Extending concurrent separation logic to the hardware level to verify the xv6 OS kernel on RISC-V with AI agents

M. Frans Kaashoek and Nickolai Zeldovich
MIT CSAIL

October 11, 2026 – revision v3*

Abstract

MachCSL is a framework for verifying system software, such as an OS kernel, on top of low-level semantics of a RISC-V computer, based on the Sail RISC-V semantics. The key idea behind MachCSL is to adapt concurrent separation logic, based on Iris, to reasoning about low-level hardware execution at the sub-instruction level: page-table translation, TLB, privilege levels, configuration registers, instruction fetch/decode/execute, traps and interrupts, DMA, shared memory, power failures, etc. Reasoning at this level of detail ensures that the system software correctly manages all of the hardware details.

Verifying software at this low level of abstraction is tedious, but LLM-based agents are capable of reasoning about such low-level details. As a case study, we verify the xv6 OS kernel (6,657 lines of C and assembly code), which provides a traditional Unix system call interface (processes, file system, file descriptors, and preemptive scheduling) and has substantial internal concurrency (multi-core support with fine-grained locking, shared memory, interrupts, DMA, etc.). In the verification process, we uncovered 11 bugs in the xv6 implementation, as well as one bug in the Sail RISC-V semantics, and one bug in Sail's Lean extraction. The verification effort took us 102 days, including the time to develop the MachCSL framework.

On top of the verified xv6 kernel, we prove a theorem about console-level behavior: e.g., if the user types `echo hello world` as input on the UART console, the only output the system can produce is `hello world` (if there are no crashes or other errors). This theorem covers both the kernel and the `init`, `sh`, `echo`, `cat`, `grep`, `sync`, and `seccomp` binaries in the initial file system image.

1 Introduction

Virtually every application uses the operating system to allocate memory, read and write files, print output, and even to initialize the system so that the application can start running. As a result, applications depend on the correctness of the operating system kernel that they are running on. Operating system kernels, however, are hard to get right. Kernels have to program the underlying hardware to correctly implement user-mode isolation for processes, such as setting up page tables, physical memory protection, and other configuration registers. Kernels have to implement sophisticated abstractions, such as processes, file systems, file descriptors, etc. Kernels also have to implement drivers that interact with peripheral devices, such as disks. Finally, kernels have significant concurrency: execution on multiple cores; interrupts taken during user-space execution and even during kernel execution itself, either from devices or from timers; DMA from devices; hardware executing page-table walks concurrently with the kernel's own accesses to the page table; and system crashes that can stop the kernel's execution at any point and start again with a fresh memory image after a reboot.

A promising approach for ensuring the absence of bugs in OS kernels is to use formal verification. Indeed, prior work has demonstrated that this can be made to work for microkernels, hypervisors, and security monitors [20, 30, 31, 34, 40, 53, 54, 66, 67, 71, 72, 74, 88, 93, 103, 108].

The contribution of this work is a machine-checked proof of the xv6 kernel [26] directly on a low-level model of the RISC-V hardware [85], which includes the semantics of virtually all hardware mechanisms needed for an OS kernel, including page tables, interrupts, etc. One advance over prior work lies in the amount of concurrency that the xv6 kernel has, and that the proof therefore has to handle: xv6 runs on multiple cores; uses spinlocks and sleeplocks; uses flag variables in shared memory for some lock-free coordination; runs with interrupts enabled in the kernel; implements preemptive context switching even for kernel code; and handles concurrent DMA from devices. xv6 also

*See §15 for a list of changes since arXiv:2609.04043v1 and v2 [51].

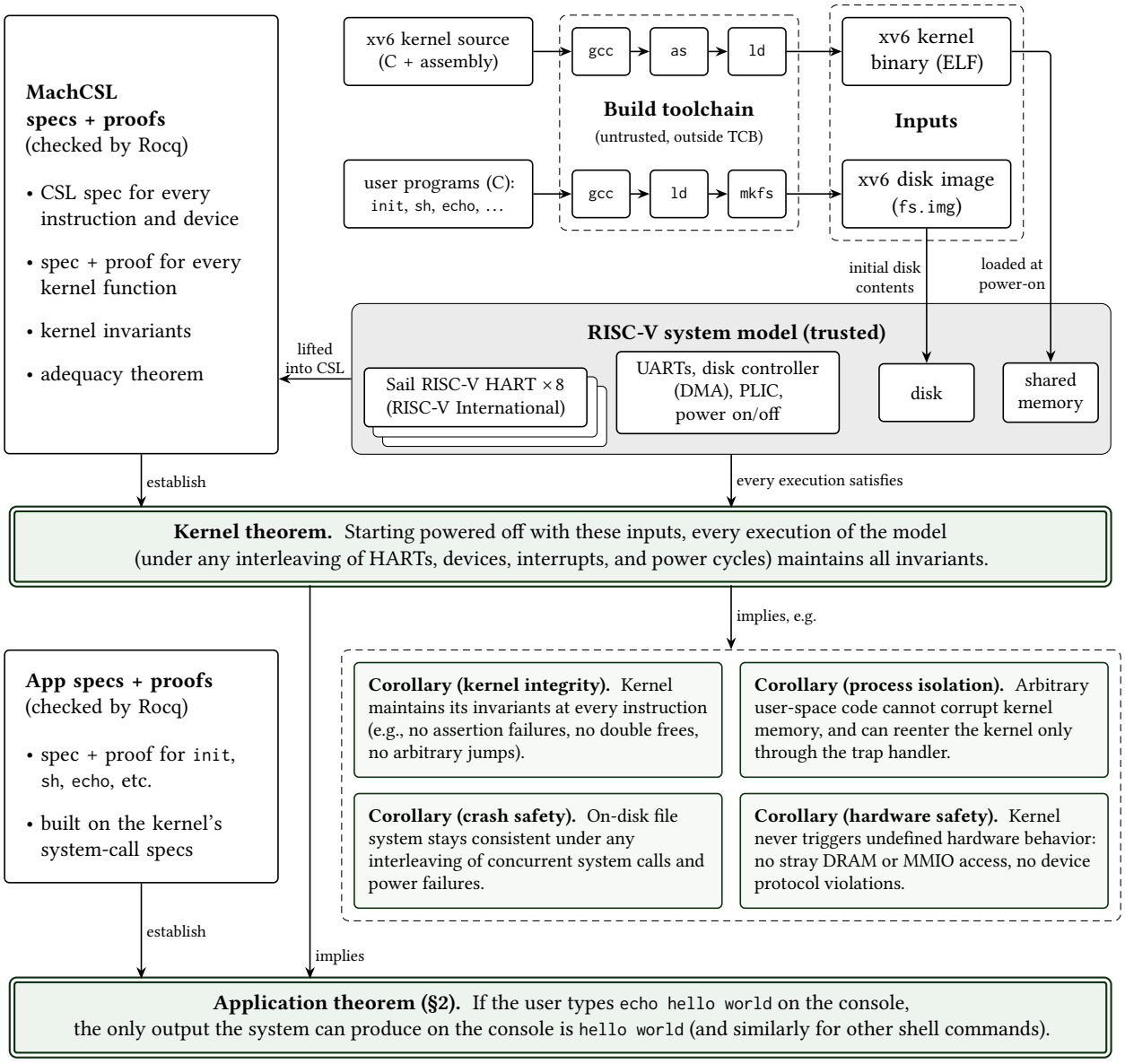

Figure 1: An overview of the proof approach for verifying the xv6 kernel, and applications running on top of it.

includes a Unix file system and system call interface, which it implements with fine-grained locking, introducing further concurrency.

Another advance is that the proof covers the interactions between the xv6 kernel and the underlying hardware by using the existing off-the-shelf semantics for the RISC-V hardware, namely the Sail model from RISC-V International [85], combined with a model of the interconnect, shared memory, devices, and power failures that we developed. This model gives a precise description of how a RISC-V core executes, down to the sub-instruction level of instruction fetch, decode, execute, and retire, along with page-table walking in hardware, TLB caching, interrupt handling, configuration registers, execution in machine/supervisor/user privilege levels, power failures and reboots, etc. By basing the xv6 kernel proof on this model, the proof covers many low-level details, including all of the boot code, all of the assembly instructions required for context switching, interrupt handling, user-kernel transitions, page-table switching, TLB handling, device DMA, spinlocks, file system crash recovery, etc.

To eliminate the entire kernel build toolchain from our verification assumptions, we verify the xv6 kernel by proving a kernel theorem about the single ELF binary produced by running `make` in the xv6 source tree, as well as the `fs.img` initial file system image produced by that same `make` command through running `mkfs` on the initial user-space

binaries. This theorem states that, in any execution of our RISC-V hardware, with the memory contents at every boot being the bytes from the kernel ELF image, and with the initial contents of the disk at first boot being the `fs.img` bytes, the execution will maintain all kernel invariants. The kernel invariants are strong enough for this kernel theorem to imply many powerful corollaries: kernel integrity, process isolation, crash safety, and hardware safety. Figure 1 shows an overview of this verification plan.

A powerful class of corollaries is application theorems that can be proven on top of the xv6 kernel's own theorem. For example, we prove a theorem about the user typing `echo hello world` on the UART serial console; the theorem states that the only output the system can produce is `hello world` (if there are no errors). This proof covers both the kernel and the user-level binaries for `init`, `sh`, `echo`, etc., that are present in the initial file system image `fs.img`. This top-level application theorem implicitly covers many aspects of kernel correctness, such as setting up page tables, recovering the file system after crashes, handling interrupts and DMA, etc.

A key challenge in realizing this verification plan is the level of detail *and* concurrency that the proof must cover. To address this challenge, this paper introduces MachCSL, which adopts ideas from concurrent separation logic (CSL), and Iris [49, 50, 60] in particular, to reason about sub-instruction-level execution of RISC-V code. CSL enables the xv6 proof to introduce abstractions that allow its proofs to stay modular, reasoning about one function, one core, or one device at a time, while handling all of the concurrency present in xv6.

Another challenge is dealing with the tedious effort of verifying the xv6 kernel at the level of the bytes in the compiled kernel image. To address this challenge, we made extensive use of AI agents (Claude Code) to help us write the proofs and intermediate specs, although we had to lay out the overall plan (MachCSL) in substantial detail, and we were heavily involved in designing intermediate layers, abstractions, and specs (e.g., our prompts to the agents added up to about 3× as much text as this paper). One pleasant benefit of using agents for proofs is that we do not have to inspect the proofs because they are machine-checked by Rocq [99], which will validate that the proofs imply the desired top-level theorem, such as our `echo` application theorem (§2).

Our main theorems and verification project have some limitations. First, our proofs cover safety properties (e.g., the kernel does not crash, and a few user-level programs that we verified run correctly), but do not yet cover liveness properties (e.g., every process will keep making progress, or the user will eventually get an output on the console), and do not cover non-interference properties (e.g., one process cannot leak data to another process or observe information from another process). Second, the theorems are subject to the assumption that our trusted computing base (TCB) is correct, which includes the Rocq proof checker and our model of the RISC-V system (which includes the Sail RISC-V core model, as well as our model of the interconnect with shared memory and peripheral devices). Third, the theorems apply to our model of shared memory: total store order (TSO) with load-load reordering. Finally, the verification relies on the top-level theorem being meaningful: what we prove is the precise theorem statement (§2), rather than some vague "xv6 is correct or bug-free".

The overall proof of the top-level theorem about the xv6 kernel and the user-level code running on top of it comprises about 1.6 million lines of Rocq code; the xv6 kernel itself, which we verified, is 6,657 lines of C and assembly, which compile down to 8,725 RISC-V instructions.[1] The proof effort took 102 days, starting from the bare idea and building the MachCSL framework from scratch, all the way to a closed theorem about the entire xv6 kernel. In the process, we discovered 11 bugs in xv6, including bugs that would crash the kernel, as well as a bug in the Sail RISC-V model and a bug in Sail's Lean extraction (§11). Other than fixing these bugs, and a few other minor changes (such as disabling unsafe code intended for use by students to debug their lab solutions that extend the xv6 kernel), the xv6 kernel that we verified is the upstream xv6 implementation, which was not designed or implemented with verification in mind.

The entire proof development is available at https://github.com/mit-pdos/xv6iris, and the precise version of the xv6 kernel covered by our theorems is available on the `verified` branch of https://github.com/mit-pdos/xv6-riscv.

2 Main correctness theorem

The main correctness theorem that captures this entire verification project is illustrated in Figure 2. This is a theorem about how the entire system will execute from the perspective of the externally visible behavior. The system includes the xv6 kernel, as well as the user-level binaries in the initial file system image (`init`, `sh`, `echo`, etc.). The externally visible behavior is just the power-on and power-off events, as well as input and output from the UART serial console. The theorem, then, is a statement about the legal traces that a system can produce.

[1] We have recently ported the entire development from Rocq to Lean, but this paper describes the Rocq-based development; all of the results described in this paper were proven in Rocq, prior to the Lean port.

Our specific theorem captures the fact that, if the user types `echo hello world` as input on the serial port, then the legal response from the system is `hello world` (or one of the possible errors, as shown in Figure 2). The theorem precisely states what the console output will be: first, `init` prints its message, then the shell prints its `$` prompt, etc. The theorem also states what errors can occur (e.g., if the kernel runs out of memory, `init` might fail to start `sh`, or `sh` might fail to start `echo`). More generally, our theorem covers a range of shell commands that a user could type, and supports several commands (`echo`, `cat`, `grep`, `sync`, and `seccomp`), together with different ways in which a user could combine them in the shell (e.g., redirecting output to a file or using pipes). The theorem also covers files that persist across power cycles: after `echo again > f` and a reboot, `cat f | grep a` must print `again`. The theorem also covers running an unverified binary under the `seccomp` sandbox (§8): after the user types `seccomp xx`, the console output is unconstrained until the next power cycle (because the sandboxed program keeps access to the console and could print arbitrary output), but the kernel remains safe and the next boot is again fully specified. Finally, the theorem does not promise what happens if the user runs other commands, such as `echo xx > /sh`, although the theorem does say that the kernel's own internal invariants continue to be maintained (i.e., the user cannot corrupt the kernel).

(a) Expected session

```
— power on —
init: starting sh
$ echo hello world
hello world
$ echo again > f
$
— power off, power on —
init: starting sh
$ cat f | grep a
again
$
```

(b) Allowed variations

```
— power on —
init: starting sh
init: exec sh failed
— power off, power on —
init: starting sh
$ echo foo
exec echo failed
$ seccomp xx
… any output …
— power off, power on —
init: starting sh
$ echo foo
foo
$
```

(c) Disallowed output

```
— power on —
init: bogus message ✗
```

```
— power on —
init: starting sh
$ echo foo
bar ✗
```

(d) Unspecified output

```
— power on —
init: starting sh
$ echo xx > /sh
… any output …
— power off, power on —
… any output …
```

Figure 2: Main correctness theorem that captures this verification project's result, illustrated through example traces that are allowed or disallowed by the theorem.

The theorem statement is powerful, because it covers many important properties of the system's execution. For example, this theorem covers all possible concurrent interleavings of multiple RISC-V cores, running the kernel code as well as the user-space processes, with preemptive scheduling of processes across cores. The theorem requires that all kernel code correctly implement system calls, interrupts, page faults, DMA for disk reads/writes, etc. The theorem requires that the kernel correctly configure page tables, correctly flush the instruction cache when setting up the code of a new user-level process on `exec`, correctly insert barriers to deal with non-sequentially-consistent shared memory, etc. This theorem also in effect requires that the kernel have no concurrency bugs, off-by-one errors, buffer overflows, use-after-free bugs, etc. The theorem covers crashes, when the system might power off at any time, and proves that the system will keep working correctly after it powers back up. This theorem also requires that the kernel safely execute arbitrary user-level code: even though the initial user binaries for `init`, `sh`, `echo`, etc., are verified, the user could also use the `seccomp` sandbox to run arbitrary unverified binaries at runtime (e.g., by running `echo xx > bin` and then running `seccomp bin`); the theorem implies that doing so will not modify any files (except through inherited file descriptors).

One limitation of this theorem is that it is a safety property: it says what outputs are legal, but it does not require the system to produce any particular output, or any output at all. However, it is easy to check that the xv6 system does actually produce output—e.g., by running it under QEMU and typing in `echo hello world`—which shows that the theorem is not vacuously true.

3 RISC-V execution model in MachCSL

To precisely reason about the execution of an OS kernel, such as xv6, MachCSL starts with a formal model of the system's execution, as shown on the left side of Figure 3 (we will explain the right side in the next two sections). Specifically, MachCSL uses the formal specification of the RISC-V ISA written in Sail, as adopted by RISC-V International [85]. The model describes the execution of a RISC-V core (more precisely, a hardware thread, or HART in RISC-V terminology), in terms of its sub-instruction-level stages, such as instruction fetch, instruction decode, execute, retire, incrementing the clock cycle, checking pending interrupts, vectoring to the trap handler, etc. The

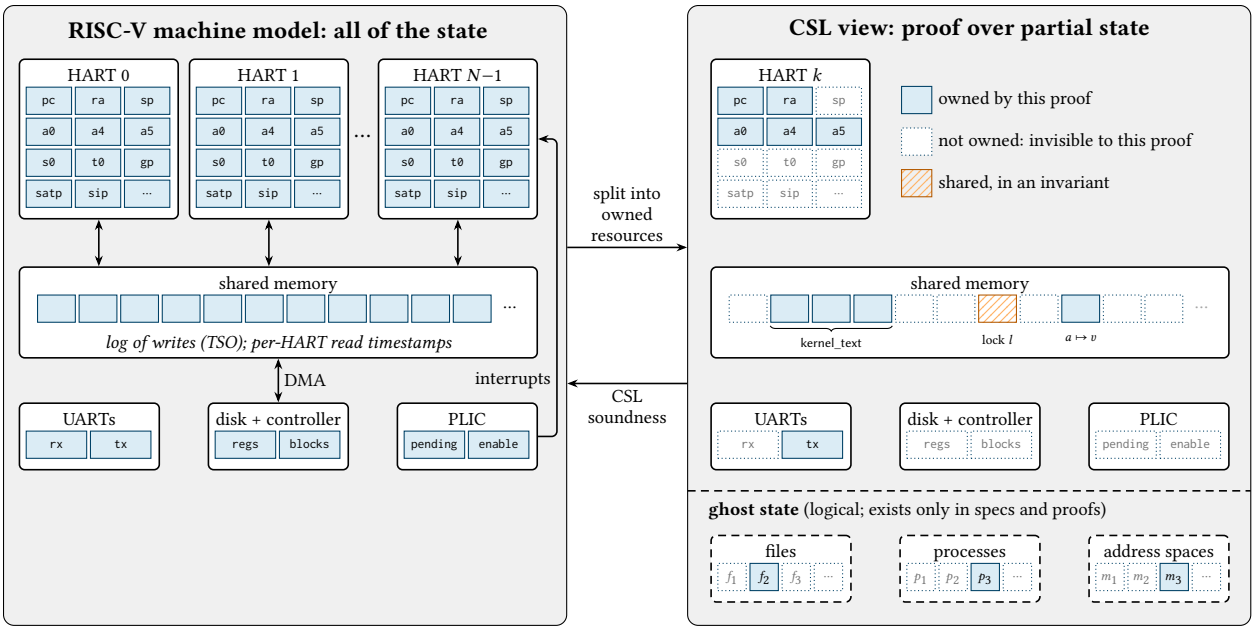

Figure 3: MachCSL's model of system execution (left, §3) and the CSL representation of it (right, §4 and §5).

model covers many detailed aspects of the CPU's execution, including the precise semantics of all control/status registers (CSRs), page tables with hardware walks, TLBs, writeback of A/D bits from the TLB into the page table, interrupt handling, user/supervisor/machine privilege levels, etc. MachCSL translates the Sail model into Rocq using Sail's Rocq backend, which generates a monadic encoding of the steps of the Sail RISC-V HART model. MachCSL uses the existing Sail RISC-V model, except for one bug that we discovered and fixed (see §11).

The HART model itself does not specify how memory works, how multiple HARTs interact through shared memory, how devices work, how DMA works, etc. Instead, the HART is defined in terms of producing various *events* throughout its execution, most notably memory-read and memory-write events, but also register-read and register-write events. To build a model of a complete RISC-V computer / SoC, MachCSL augments the Sail HART model with a model of shared memory; a model of several devices, including two UARTs and a DMA-capable disk controller; and a model of the PLIC interrupt controller. All of these components (multiple HARTs, the devices, and the interrupt controller) execute concurrently, with fine-grained interleaving at the level of individual events.

Interleaving at the level of individual memory reads and writes allows MachCSL's model to capture fine-grained concurrency, such as a hardware page-table walk interleaving with another core updating the page table, or another core doing writeback of A/D bits to the same page-table entry. Interleaving at the level of register reads and writes allows MachCSL to model the asynchronous delivery of interrupts into the pending-interrupt status register of individual HARTs, which can occur at any instant. In particular, MachCSL's fine-grained interleaving captures not just interleaving at the level of instructions, but interleaving at the level of sub-instruction steps, such as individual memory reads during the page-table walk, instruction fetch, data fetch, etc.

MachCSL's shared-memory model is TSO with load-load reordering: it captures TSO semantics for writes, which allows CPUs to implement write buffering, and load-load reordering for reads, which allows CPUs to do out-of-order loads (unlike x86-TSO [89]). In particular, the memory state is a log of write events, where each event specifies the address and value of the write, and the source of the write (a specific core, or a specific device issuing DMA writes). Writing to memory appends a new event to this log. Reads under TSO can be stale: each core maintains a timestamp (log position) that represents the prefix of the log that must be visible to the core for purposes of memory reads. Thus, when a core executes a memory read, it picks some log position at or after that core's timestamp, and gets the latest value written to that address as of that position. There are two exceptions to this rule: first, cores see their own writes (i.e., they check their own store buffer), and second, cores have read coherence (i.e., once a read of some address returns a value from a certain timestamp, future reads of that address do not roll back to earlier timestamps). Atomic instructions, modeled in Sail as exclusive memory reads and writes, force the core to increase its timestamp to the end of the global log, which means the atomic instruction will access the latest state. The RISC-V `fence` instruction names

the kinds of accesses (reads and/or writes) before and after it that must be ordered. A write-to-read `fence` advances the core's timestamp to its own latest write (to any address). A read-to-read `fence` advances the core's timestamp to the highest read timestamp observed so far (at any address).

RISC-V allows instruction caches to be non-coherent with data accesses. To represent this, our model includes a second timestamp associated with every core; this timestamp determines what data can be returned in response to an instruction fetch. Each instruction fetch non-deterministically reads at any log position at or after the core's instruction timestamp, and does not advance the timestamp. The `fence.i` instruction bumps up the instruction timestamp to the core's latest write (or to its data timestamp, if that is later).

MachCSL also models power failures and reboots. It does so using a logical "thread" of execution that invokes two operations in a loop: power-on and power-off. The power-on operation boots up the system, and starts executing eight concurrent HARTs (xv6's `NCPU`), representing all of the cores. The HARTs start running at the initial PC as specified by the platform (`0x80000000`, the start of DRAM on QEMU's `virt` machine), in machine (M) mode. The initial memory contents at power-on are the bytes from the xv6 kernel ELF image, as generated by running `make` on the xv6 kernel's source code; there is no difference between bytes representing code and data. The initial registers on each HART are set non-deterministically, except where mandated by the RISC-V specification or by platform assumptions that the xv6 kernel makes (e.g., specific bits in the `misa` register, or specific features supported by the HART). When the system powers off, all existing HARTs halt and do not execute any more instructions. The next power-on creates new logical HARTs, representing the next era of execution, with fresh memory, fresh initial register contents, etc.

The persistent on-disk state is logically preserved across power cycles; the power-on operation starts the disk device with the state from the previous power-off. At the initial power-on, the disk contains the byte-level image of `fs.img` as generated by `mkfs` as part of running `make` in the xv6 source directory, which builds all of the user-space programs, initializes an empty file system image, and writes the compiled user-space binaries to that image. The disk model captures the atomic-sector-write assumption, which file systems traditionally rely on for crash safety (though the model allows larger writes to be torn across sector boundaries), and models writeback caching in the disk controller (although xv6 does not take advantage of it).

Finally, the system model produces a trace of events, which includes the power-on and power-off events, as well as bytes sent and received on the UART serial ports. This trace allows stating a succinct theorem about the externally visible aspects of the execution of the system, such as what the user will see on the console, depending on what they type as input to the console; one example is the `echo` application theorem from §2.

4 MachCSL by example

The RISC-V model described above captures a large number of possible interleavings. For example, at any given moment, any core (HART) could take a "step": load or store a register, fetch some page-table entries from memory, load some instruction bytes, load or store data accessed by the instruction, update the TLB, etc. Concurrently, devices can also "step": either perform I/O such as reading or writing to the durable disk, or perform memory reads or writes through DMA, or even raise interrupt flags in some core's interrupt status register. Directly reasoning about these interleavings—such as explicitly considering all of these different orders of events in the proof—is cumbersome.

Concurrent separation logic (CSL) makes it feasible to prove correctness under all of the interleavings that can occur in the system, but to do so by proving correctness of "sequential" executions. That is, the proof can step through the execution of code one instruction at a time on a particular core, rather than having to step through all of the system-wide interleavings across cores.

To achieve this, CSL requires reasoning about each individual core in a different execution model that, instead of directly accessing memory, registers, etc., explicitly tracks ownership of state (e.g., memory and registers) and other resources, as shown on the right side of Figure 3. The CSL framework then proves that the rules for reasoning about instructions in this CSL model of the machine imply a theorem statement about the execution of the same instructions in the underlying highly concurrent execution model (e.g., the RISC-V machine model from §3), illustrated by the "CSL soundness" arrow in Figure 3.

Figure 4 shows an example of the benefit of the CSL approach, illustrating how CSL can be used to prove a simple function `f` that acquires a lock in order to increment a shared counter. The proof of the overall function is built on top of proofs for its three constituent pieces: calling `acquire`, incrementing `count`, and calling `release`, as if executing sequentially. Each of these three blocks requires ownership of some subset of resources (e.g., `acquire` requires ownership of the CPU registers touched by its implementation; the body requires ownership of the `count` variable in memory; and all of the pieces require knowing that the kernel text instructions are present in memory).

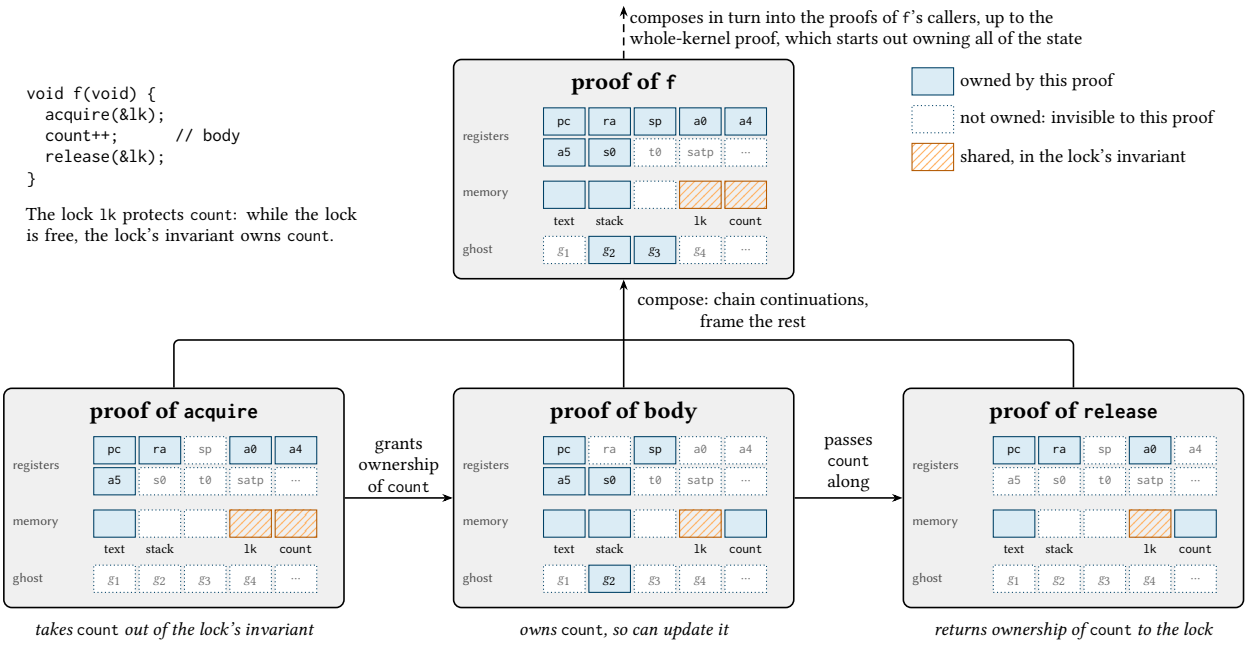

Figure 4: A simplified view of how proofs of functions compose in CSL.

The proof of the overall function, then, requires ownership of all of the resources needed for the combination of those three blocks.

The key benefit of CSL in this example is that the proof of `f` ensures that `f` executes correctly, even though the proof never had to explicitly consider the possibility of another core trying to acquire the lock while `f` runs, an interrupt arriving just as `acquire` starts executing, devices performing DMA accesses to memory, etc. The correctness of all these concurrent events is a consequence of all of the proofs (e.g., concurrent `acquire` calls, interrupt handlers, device proofs, etc.) following the same consistent set of CSL rules, which make it possible for a top-level adequacy theorem to compose all of these individual proofs into an overall whole-system theorem that covers all possible interleavings.

4.1 Spinlock example

To explain how MachCSL uses CSL to reason about kernel execution at the RISC-V instruction level, this section will present a simple implementation of a spinlock as an example (the `acquire` and `release` functions that showed up in Figure 4), along with the rules that CSL provides for reasoning about such code. MachCSL in particular builds on top of the Iris concurrent separation logic [49, 50, 60], as mechanized in Rocq [99]. The C code of the example spinlock implementation that we will consider is shown in Figure 5. The simplified example we present in this section assumes a sequentially consistent memory model.

All MachCSL specifications and proofs are built on top of the RISC-V hardware model described in the previous section. To specify and reason about this C implementation, the developer first compiles the C code into machine code; the resulting assembly for this spinlock implementation is shown in Figure 6 (the `fence` instruction in `release` matters only under a weak memory model, so this section ignores it). Grounding all specifications and proofs in machine code allows MachCSL to factor out the compiler, linker, assembler, and other parts of the build process from the trusted computing base; the theorems proven in MachCSL are statements about executing the machine code on the RISC-V model, which avoids reasoning about C semantics, compiler correctness, build flags, etc.

4.2 Resources

MachCSL specifications are formulated in terms of concurrent separation logic. A key idea in concurrent separation logic is the notion of ownership of resources. For instance, a specification can represent ownership of memory location a using a resource $a \mapsto_{\mathsf{Mem}} v$, which says two things: first, memory location a contains value v, and second, no other thread owns this resource. The second point implies that no other thread can read or write memory location a, and therefore, the proof can reason about its own accesses to a without worrying about concurrency. As an analogy, ownership of $a \mapsto_{\mathsf{Mem}} v$ is similar to Rust's notion of some function owning a mutable reference to a. Much as in

```c
struct spinlock {
  int locked; // Is the lock held?
};

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

void
release(struct spinlock *lk)
{
  __atomic_store_n(&lk->locked, 0, __ATOMIC_RELEASE);
}
```

Figure 5: Simplified C implementation of a spinlock in the xv6 kernel.

```
acquire:
    li  a4,1

.L2:
    mv  a5,a4
    amoswap.w.aq    a5,a5,(a0)
    sext.w  a5,a5
    bnez    a5,.L2
    ret

release:
    fence   rw,w
    sw  zero,0(a0)
    ret
```

Figure 6: Compiled assembly of the spinlock from Figure 5.

separation logic, Rust's borrow checker guarantees that if one function holds a mutable reference, no other function can hold a mutable or shared reference to the same variable.

The specification for executing code, such as the spinlock `acquire` and `release` functions, is stated in terms of separation-logic implication—that is, defining what resources are required to invoke the function. For example, Figure 7 shows the specification for the example spinlock. The $-\!\!*$ operator used in this specification is separation logic's implication: it says that, given resources on the left of the $-\!\!*$, a proof can obtain the resources on the right of the $-\!\!*$.

The $-\!\!*$ operators chain, so the specification of `acquire`, for example, requires the caller to establish all of the facts required by the chain of $-\!\!*$ operators to invoke `acquire`. `pc` $\mapsto_{\text{Reg}}$ `acquire` is a requirement that the `pc` register contain the address of `acquire`, which is where `acquire`'s first instruction lives, as shown in Figure 6. `a0` $\mapsto_{\text{Reg}}$ l requires that the `a0` register—the first argument in the RISC-V calling convention—contain the address of the lock, l. The next two preconditions require passing ownership of both the `a4` and `a5` registers, containing any value; `acquire` does not care what the existing values are, but it will overwrite them. The `ra` $\mapsto_{\text{Reg}}$ r resource passes ownership of the return-address `ra` register, and says that it contains some return address r.

Resources can be more general than just ownership of specific registers or memory locations. For example, kernel_text is a resource representing the kernel's text section (instructions) in memory (which happens to include the code of `acquire` and `release`). This kernel_text resource is non-exclusive ("persistent"), meaning that multiple proofs (e.g., concurrent HARTs) can own it at the same time, because it states that the kernel text is read-only. As a result, this resource is sufficient as a precondition for reading memory (during instruction fetch), but is not sufficient as a precondition for memory write instructions. Thus, it would not be possible to prove correctness of code that performs a store into the kernel's text. The other two resources in this specification, is_lock and the parenthesized continuation, are also more than just simple memory ownership, and we will discuss them later in this section.

$\texttt{pc} \mapsto_{\text{Reg}} \texttt{acquire} \mathrel{-\!\!*}$
$\texttt{a0} \mapsto_{\text{Reg}} l \mathrel{-\!\!*}$
$(\exists\, v_4.\ \texttt{a4} \mapsto_{\text{Reg}} v_4) \mathrel{-\!\!*}$
$(\exists\, v_5.\ \texttt{a5} \mapsto_{\text{Reg}} v_5) \mathrel{-\!\!*}$
$\texttt{ra} \mapsto_{\text{Reg}} r \mathrel{-\!\!*}$
kernel_text $\mathrel{-\!\!*}$
is_lock $l\ P \mathrel{-\!\!*}$
($\texttt{pc} \mapsto_{\text{Reg}} r \mathrel{-\!\!*}$
  $\texttt{a0} \mapsto_{\text{Reg}} l \mathrel{-\!\!*}$
  $(\exists\, v_4.\ \texttt{a4} \mapsto_{\text{Reg}} v_4) \mathrel{-\!\!*}$
  $(\exists\, v_5.\ \texttt{a5} \mapsto_{\text{Reg}} v_5) \mathrel{-\!\!*}$
  $\texttt{ra} \mapsto_{\text{Reg}} r \mathrel{-\!\!*}$
  $P \mathrel{-\!\!*}$
  wp CpuLoop) $\mathrel{-\!\!*}$
wp CpuLoop

(a) Specification for `acquire`.

$\texttt{pc} \mapsto_{\text{Reg}} \texttt{release} \mathrel{-\!\!*}$
$\texttt{a0} \mapsto_{\text{Reg}} l \mathrel{-\!\!*}$
$\texttt{ra} \mapsto_{\text{Reg}} r \mathrel{-\!\!*}$
kernel_text $\mathrel{-\!\!*}$
is_lock $l\ P \mathrel{-\!\!*}$
$P \mathrel{-\!\!*}$
($\texttt{pc} \mapsto_{\text{Reg}} r \mathrel{-\!\!*}$
  $\texttt{a0} \mapsto_{\text{Reg}} l \mathrel{-\!\!*}$
  $\texttt{ra} \mapsto_{\text{Reg}} r \mathrel{-\!\!*}$
  wp CpuLoop) $\mathrel{-\!\!*}$
wp CpuLoop

(b) Specification for `release`.

Figure 7: Specifications for the example spinlock in Figure 6.

4.3 Reasoning about machine execution

One important aspect of MachCSL lies in how it uses CSL-style resources to specify the execution of code in a system. In a traditional Hoare-logic style, specifications take the form of pre- and postconditions associated with some function. In this style, one might expect that there is a clear pre- and postcondition associated with running `acquire`.

Having explicit pre- and postconditions in a specification implicitly requires that the execution semantics be defined in a compatible way, in terms of executing functions, where functions eventually return to their callers. This is not how hardware semantics are defined: there is no notion of functions, or returning to a caller, and there is not even a notion of the PC necessarily advancing sequentially through code. For instance, at any time, an interrupt might come in and redirect the CPU's execution to the interrupt handler, rather than advancing the PC to the next instruction. Moreover, even if an interrupt does not come in, the CPU will fetch the next instruction based on the state of the TLB, page table, and the physical memory at the PC address, which might not even be a valid DRAM address; all these are facts that the proof must explicitly consider.

To address this, all specifications in MachCSL are ultimately statements about the low-level execution of the RISC-V CPU, which is defined based on the Sail semantics: the CPU keeps fetching instructions from the PC (through virtual memory translation), decoding them, executing them, retiring them, possibly vectoring to the interrupt handler, etc., in an infinite loop. We formally capture the notion that it is safe to run the CPU in this way using the resource wp CpuLoop.[2] Here, wp stands for "weakest precondition", meaning that wp CpuLoop is a resource that is sufficient to satisfy the precondition of running CpuLoop, which is our infinite loop of RISC-V CPU execution.

One implication of the wp CpuLoop encoding is that there are no postconditions, because the CPU is never done. Instead, wp CpuLoop says that the CPU can start running, and from that point onwards, it can continue running safely. To illustrate what wp CpuLoop means, consider a simple example where the CPU is stuck in an infinite loop: the PC points at an instruction that jumps back to the same exact PC value, and there is nothing that could possibly interrupt the CPU (no interrupts enabled, no shared page tables, etc.). In this case, a proof can directly establish that wp CpuLoop holds, by inductively arguing that the CPU will forever keep executing this one instruction safely.

In the general case, MachCSL views the safety of the xv6 kernel in a similar way: it effectively treats the OS kernel, along with any user-space programs it executes, interrupts it takes, etc., as one big complicated infinite loop. The kernel boots up, initializes its state, and then enters an infinite loop of running user-space processes, switching page tables, handling system calls, handling interrupts, context-switching between processes, etc. Thus, proving the safety of the xv6 kernel's implementation requires establishing wp CpuLoop at the initial CPU starting configuration—that is, it is safe to enter this "infinite loop" starting from whatever state the CPU happens to have when it first powers up.

[2]"Safe" here means that the execution proceeds based on the underlying operational semantics, maintains all invariants, and does not trigger any undefined behavior. This is a somewhat technical statement, but it turns out to allow the MachCSL proof to conclude an adequacy theorem that implies strong corollaries about the xv6 kernel, as illustrated in Figure 1.

The remaining challenge lies in proving this "infinite loop" wp CpuLoop statement for the entire kernel. We do not want the proof to consider this entire complex loop all at once; that would not be feasible. Instead, we want to decompose the proof along modular abstraction boundaries, such as function calls and interrupt handlers: for instance, the proof of a function like `acquire` should be an argument about the execution of that function's instructions. What makes this decomposition tricky is that a proof about a single function does not know why (or whether) it is going to be safe to continue executing once the function returns. For example, the proof of `acquire` does not know whether, after `acquire` jumps back to the return address in the `ra` register, the caller's code might write garbage into kernel memory in a way that violates the kernel's invariants.

To enable this decomposition, most specifications in MachCSL are formulated in terms of *continuations*. For example, the specification of `acquire`, shown in Figure 7, says that wp CpuLoop holds, as long as the caller satisfies the precondition, and also supplies a proof about "the rest of the infinite loop" when `acquire` returns, represented by the last premise of the specification. That last premise says that it is the job of the caller's proof to establish wp CpuLoop starting from the state at which `acquire` is going to return. In this state, the PC will be at r, which is the return-address register's contents; ownership of the `a0`, `a4`, `a5`, and `ra` registers is given to the continuation; and the continuation also receives the P predicate that is the result of acquiring the lock (we will describe this P predicate shortly).

This formulation allows `acquire`'s proof to establish that it will be safe to forever execute the machine: `acquire` itself maintains safety while executing `acquire`'s instructions, and the caller proved that it will continue to be safe to execute after `acquire` returns. In this example, there is just one continuation shown, for returning to `acquire`'s caller, but in practice in xv6 proofs there are often multiple continuations (some hidden inside of abstract predicates), such as a continuation for proving the safety of jumping to the interrupt handler at any point (instead of continuing on to execute the current PC), a continuation for the safety of context-switching to another kernel thread, etc.

4.4 Lock specification

Returning to the P predicate, when `acquire` returns, the caller can assume that the lock has been acquired, which in CSL is represented by gaining ownership of some predicate P associated with the lock. The is_lock l P premise in `acquire`'s spec ties the predicate P to the lock address l that is being acquired. The specification of the spinlock allows the developer to associate any predicate P with the spinlock. For instance, this P likely contains memory points-to facts for shared memory locations protected by this lock.

In our shared-counter example from Figure 4, the P predicate would be $\exists\, v.\ \texttt{count} \mapsto_{\mathsf{Mem}} v$, granting ownership of the `count` variable in memory, with some value v. Owning $\texttt{count} \mapsto_{\mathsf{Mem}} v$ allows the body of `f` to assume exclusive ownership of that memory location; the CSL rules mean that no other HART can be reading or writing that memory location at the same time. As a result, the proof of `f`'s body can proceed without having to consider the possibility of concurrent accesses to the same memory. In particular, `count++` could be compiled down to three non-atomic instructions (load, increment, and store) and the proof need not worry about the possibility of another HART modifying the memory between the load and store.

The specification for `release` does the opposite operation with P: it is now the caller's obligation to return ownership of P to the spinlock in order to release it. This ensures that the predicate P holds for the next lock holder's acquisition. In the shared-counter example, calling `release` requires the proof to return ownership of $\exists\, v.\ \texttt{count} \mapsto_{\mathsf{Mem}} v$.

4.5 Instruction specifications

To prove the correctness of the spinlock implementation, a proof in MachCSL must reason about the individual instructions that make up the implementation, chaining them together to conclude the specification of the entire `acquire` or `release` function. For example, Figure 8 shows specifications for several individual RISC-V instructions that appear in `acquire`'s machine code. Each of these specifications requires the caller to prove the premises of the spec based on the resources the caller already holds.

For instance, the first instruction of `acquire` is a load-immediate (`LI`). The proof of `acquire` uses the specification of the `LI` instruction shown in Figure 8(a). The `acquire` proof holds the resource $\texttt{pc} \mapsto_{\mathsf{Reg}} \texttt{acquire}$, as well as the resource $\texttt{a4} \mapsto_{\mathsf{Reg}} v_4$, which it uses to satisfy `LI`'s spec. The instruction specification also requires the caller to prove that this instruction is actually present in memory at the PC's address (the instr resource). This fact is implied by kernel_text, which is in the precondition of `acquire`. The only remaining fact that `acquire`'s proof has to supply in order to use `LI`'s specification is the continuation. This effectively reduces `acquire`'s remaining proof obligation to proving a wp CpuLoop while holding $\texttt{pc} \mapsto_{\mathsf{Reg}} (\texttt{acquire} + 2)$ as well as $\texttt{a4} \mapsto_{\mathsf{Reg}} 1$. This allows the proof of `acquire` to keep stepping through its instructions until it hits the return instruction, `RET`. The specification of `RET` requires

$$\begin{array}{l}\mathsf{pc} \mapsto_{\mathsf{Reg}} i \mathrel{-\!\!*} \\ \mathsf{instr}\, i\ (\mathtt{LI}\ r, x) \mathrel{-\!\!*} \\ (\exists\, v.\ r \mapsto_{\mathsf{Reg}} v) \mathrel{-\!\!*} \\ (\ \mathsf{pc} \mapsto_{\mathsf{Reg}} (i+2) \mathrel{-\!\!*} \\ \quad r \mapsto_{\mathsf{Reg}} x \mathrel{-\!\!*} \\ \quad \mathsf{wp\ CpuLoop}\ ) \mathrel{-\!\!*} \\ \mathsf{wp\ CpuLoop}\end{array}$$

(a) Specification for LI.

$$\begin{array}{l}\mathsf{pc} \mapsto_{\mathsf{Reg}} i \mathrel{-\!\!*} \\ \mathsf{instr}\, i\ (\mathtt{RET}) \mathrel{-\!\!*} \\ \mathsf{ra} \mapsto_{\mathsf{Reg}} r \mathrel{-\!\!*} \\ (\ \mathsf{pc} \mapsto_{\mathsf{Reg}} r \mathrel{-\!\!*} \\ \quad \mathsf{ra} \mapsto_{\mathsf{Reg}} r \mathrel{-\!\!*} \\ \quad \mathsf{wp\ CpuLoop}\ ) \mathrel{-\!\!*} \\ \mathsf{wp\ CpuLoop}\end{array}$$

(b) Specification for RET.

$$\begin{array}{l}\mathsf{pc} \mapsto_{\mathsf{Reg}} i \mathrel{-\!\!*} \\ \mathsf{instr}\, i\ (\mathtt{AMOSWAP}\ r_v, r_v, (r_a)) \mathrel{-\!\!*} \\ r_a \mapsto_{\mathsf{Reg}} a \mathrel{-\!\!*} \\ r_v \mapsto_{\mathsf{Reg}} v \mathrel{-\!\!*} \\ (\mathsf{True} \Rrightarrow \exists\, x.\ a \mapsto_{\mathsf{Mem}} x * (a \mapsto_{\mathsf{Mem}} v \Rrightarrow Q\ v\ x)) \mathrel{-\!\!*} \\ (\ \mathsf{pc} \mapsto_{\mathsf{Reg}} (i+4) \mathrel{-\!\!*} \\ \quad r_a \mapsto_{\mathsf{Reg}} a \mathrel{-\!\!*} \\ \quad r_v \mapsto_{\mathsf{Reg}} x \mathrel{-\!\!*} \\ \quad Q\ v\ x \mathrel{-\!\!*} \\ \quad \mathsf{wp\ CpuLoop}\ ) \mathrel{-\!\!*} \\ \mathsf{wp\ CpuLoop}\end{array}$$

(c) Specification for AMOSWAP.

Figure 8: Specifications for individual RISC-V instructions.

proving a wp CpuLoop for $\mathsf{pc} \mapsto_{\mathsf{Reg}} r$, where r is the return address in the ra register. The proof of acquire already has this wp CpuLoop, provided by acquire's caller, which allows the proof to conclude.

4.6 Framing ownership

The specifications for individual instructions, such as LI and AMOSWAP, mention ownership of specific registers and memory locations, but say nothing about ownership of other registers and the rest of the memory. This is a key aspect of CSL's approach to modularity. If consumers of these specifications (e.g., proofs that use the specs of LI and AMOSWAP) own some resources that are disjoint from the ones mentioned in the specification, those consumers can *frame* their ownership over the invocation of this spec.

For example, consider the proof of acquire. The proof starts out by owning many resources, stemming from the precondition of acquire's specification:

$$\mathsf{pc} \mapsto_{\mathsf{Reg}} \mathsf{acquire} * \mathsf{a0} \mapsto_{\mathsf{Reg}} l * \mathsf{a4} \mapsto_{\mathsf{Reg}} v_4 * \mathsf{a5} \mapsto_{\mathsf{Reg}} v_5 * \mathsf{ra} \mapsto_{\mathsf{Reg}} r * \mathsf{kernel_text} * \mathsf{is_lock}\ l\ P$$

When the proof invokes the specification of LI, which is the first instruction of acquire, it chooses the i variable in LI's spec to be the address acquire, which indeed contains the LI instruction. The proof then derives the $\mathsf{instr}\, i\ (\mathtt{LI}\ r, x)$ fact from kernel_text, with $i = \mathsf{acquire}$, $r = \mathsf{a4}$, and $x = 1$. The proof then passes ownership of $\mathsf{instr}\, i\ (\mathtt{LI}\ r, x)$, $\mathsf{pc} \mapsto_{\mathsf{Reg}} i$, and $\mathsf{a4} \mapsto_{\mathsf{Reg}} v_4$ to LI's specification. The remaining resources held by acquire's proof at this point get passed to the continuation. The continuation's proof gets to combine these passed resources (unchanged) with the resources returned from LI, which have been updated. In particular, the resources handed back by LI's specification include $\mathsf{pc} \mapsto_{\mathsf{Reg}} (i+2)$ and $\mathsf{a4} \mapsto_{\mathsf{Reg}} 1$, which allows the rest of the proof to proceed. The key modularity property here is that LI's specification and proof never had to mention these other resources, or even know what kinds of other resources might exist.

Framing also enables composition at the level of function proofs. In the shared-counter example from Figure 4, the proof of f owns the s0 register, and uses framing to preserve ownership of s0 across its calls to acquire and release (which do not require ownership of that particular resource).

4.7 Invariants

Memory locations that are accessed by multiple threads, such as the spinlock's lock bit itself, cannot be owned by any one thread, because all threads that are trying to acquire the spinlock need to be able to access it at the same time. In CSL, this is represented using the notion of an invariant, which is a separation-logic resource (predicate) that is true at all times. In the example from Figure 4, the lk and count memory locations are owned by the lock invariant, as indicated by the orange-striped cells; when the lock is acquired, ownership of count passes to the proof of the function that acquired the lock.

For example, the is_lock resource that showed up in the specifications of acquire and release is precisely such an invariant, and its definition is shown in Figure 9, where the box denotes an invariant. The definition says that, at every instant, the lock invariant owns the memory location l, and it has some value v. Moreover, if v is 0, the lock invariant also owns the lock predicate P. Otherwise, v must be 1, and there is no P in the lock invariant (it has been handed out to the acquiring thread). Like kernel_text, an invariant is persistent: any number of proofs can

hold is_lock $l\ P$ at the same time. This is why the specifications of `acquire` and `release` do not return kernel_text or is_lock $l\ P$ to their continuations; the caller simply keeps its own copy.

$$\text{is_lock}\ l\ P \triangleq \boxed{\exists\, v.\ \ l \mapsto_{\text{Mem}} v * ((v = 0 * P) \vee (v = 1))}$$

Figure 9: Invariant for the spinlock example.

Interacting with the invariant requires a special kind of specification for an instruction. Instead of passing in full ownership of the memory location in the precondition (which would preclude concurrent access by multiple threads), the specification of the atomic `AMOSWAP` instruction, as shown in Figure 8(c), allows the caller to open the invariant for an instant during the instruction's execution. This is encoded in `AMOSWAP`'s precondition using the $\Rrightarrow$ resource, as we will now explain.

The $\Rrightarrow$ notation encodes a proof callback (in Iris, this object is called a view shift, and is defined in terms of the update modality or fancy-update modality). Specifically, $A \Rrightarrow B$ is a resource that allows the holder to "invoke" the callback by giving up ownership of A, and receive ownership of B in return; a callback that requires no resources to invoke is written $\text{True} \Rrightarrow B$. The precondition of `AMOSWAP` uses $\Rrightarrow$ to encode the fact that, at some instant during `AMOSWAP`'s execution, the caller will be required to give ownership of $a \mapsto_{\text{Mem}} x$, for some value x, and also give another $\Rrightarrow$ resource: a proof callback that, given $a \mapsto_{\text{Mem}} v$, returns ownership of some resource $Q\ v\ x$, where Q is specified by the caller of this specification. Not shown in the specification is the coupling between the two $\Rrightarrow$ callbacks, requiring them to be executed together at the same point in the proof (invoking the outer $\Rrightarrow$ obligates the invoker to also invoke the inner $\Rrightarrow$ before continuing with more execution steps).

This callback machinery allows the caller to open an invariant, such as is_lock, to temporarily obtain ownership of resources stored in the invariant, for an instantaneous point in time. The outer $\Rrightarrow$ corresponds to allowing the caller to open the invariant, and the inner $\Rrightarrow$ allows the caller to close the invariant back up with a new value stored at the lock's memory address. The Q predicate allows the caller to pass along any resources that it may have taken out of the invariant while it had it open.

For example, in the proof of `acquire`'s invocation of `AMOSWAP`, the $Q\ v\ x$ predicate is defined as holding no resources if x is 1, and holding P if x is 0. In `acquire`'s use of `AMOSWAP`, the v argument to Q does not matter. Q encodes what must be true when `AMOSWAP` finishes: if the atomically exchanged old value was 1, then `acquire` did not acquire the lock, and as a result, it got no resources. But, if `AMOSWAP` observed that the old value was 0 (and it was replaced by 1), then the caller acquired the lock, and Q contains P. Specifically, once the outer $\Rrightarrow$ callback opens the lock invariant and gives up $a \mapsto_{\text{Mem}} x$, and the inner callback receives $a \mapsto_{\text{Mem}} v$ as the new contents of the memory location, the proof can consider two possible cases: either x was 0 or it was 1. If x was 1, this means the lock was already held, and Q is empty. On the other hand, if x was 0, then the lock invariant previously held P, but now does not need to hold P, because the location is now 1. This allows the callback to keep P in Q and reclose the invariant without P.

4.8 Discussion

An advantage of MachCSL is that it reasons about the execution of code based on precise semantics for RISC-V hardware. This is important for an OS kernel, because the OS kernel is responsible for setting up the hardware mechanisms needed to ensure its own correctness, such as configuring page tables, managing the TLB, setting configuration registers, setting up interrupt trap handlers, enabling and disabling interrupts, etc. MachCSL makes reasoning about all of these operations explicit and consistent with reasoning about all other parts of the kernel's implementation. This allows a MachCSL proof to make relatively few assumptions about the kernel's correctness: we do not need to assume that memory accesses are properly synchronized, that interrupts are properly handled, that page tables are correctly configured, that the kernel correctly flushes the TLB when switching between user and kernel page tables, etc.

One cost of MachCSL is that it requires the proof to be explicit about the execution of each instruction, and to consider many details. This level of tedium would be prohibitive if the proofs (and even the intermediate specs for each function inside xv6) had to be written by hand. Using agents makes it possible to automate large parts of the proof effort required to verify xv6 in MachCSL.

A second cost of MachCSL is that the proof is tied specifically to the precise binary image of the kernel. As a result, small changes to the kernel's source code can lead to pervasive changes throughout the kernel image (such as shifting relative offsets or instruction alignments), thereby requiring many proofs to be fixed. This again is a tedious task that would be impractical to do by hand at each kernel recompile, yet we find that agents are able to perform these tedious proof updates without substantial developer input.

Although MachCSL is based on low-level RISC-V semantics, higher-level proofs need not reason directly about the precise bit-level representation of all system state. Instead, developers introduce abstractions and invariants that connect the low-level machine state, such as registers and memory, to higher-level abstract states and predicates. For example, the xv6 proof introduces abstractions such as a logical mapping from virtual to physical addresses (abstracting away the 3-level page-table tree structure and the TLB); a file on disk containing some sequence of bytes; a runnable process with a PID and name; etc. Furthermore, CSL allows splitting up ownership of these abstractions at the level of abstract state, rather than physical state. That is, a proof of some function, such as `sys_write`, can own the bytes of a file, regardless of whether those bytes are currently stored in the buffer cache, in the write-ahead log, or installed in the file inode's actual data blocks. This is implemented by introducing *ghost state*, which is logical state that appears only in the specs and proofs (as shown in Figure 3 and Figure 4). Ghost state can be connected to physical state through explicit predicates and invariants, and ghost state can then be owned by different threads much like physical resources such as registers and memory locations.

Finally, MachCSL inherits the key benefit of CSL-style local reasoning. In our example spinlock specification and proof, CSL allows the caller to reason about its own execution as if no other threads could interfere: the caller gains complete ownership of the locked state, as described by P, and the proof implicitly promises that no other CPUs can modify memory in a way that would invalidate P, that no concurrent device DMA could occur that would overwrite P's memory, and that no interrupts can happen that could violate P. This allows the formal proof to avoid explicit reasoning about all of these possible interleavings. At the same time, the theorem proven using CSL does establish the fact that the kernel is correct for all of these possible interleavings.

5 CSL for RISC-V

This section explains how MachCSL provides reasoning principles for the Sail RISC-V model using CSL.

5.1 Ownership of state

CPU state. MachCSL represents the ownership of every CPU register using a points-to fact, $r \mapsto_{\mathsf{Reg}} v$. Each HART has its own set of registers, so the points-to resource is indexed by the HART ID (we omit this index for simplicity in this paper's presentation). The register set includes the 32 general-purpose RISC-V registers, which showed up in the previous section's spinlock example, as well as many control and status registers, which the Sail model folds into the notion of a "register". For example, the TLB is modeled as a separate register in the Sail model, storing all of the TLB entries; MachCSL represents it as ownership of $\texttt{tlb} \mapsto_{\mathsf{Reg}} \textit{tlb-entries}$. The RISC-V PMA and PMP tables are also represented by registers. The `satp` and `stvec` registers, storing the page-table root pointer and the trap vector, are similarly represented.

Representing ownership of each register using a separate points-to resource provides two advantages. First, it allows MachCSL to accurately model registers that are concurrently modified by other parts of the system, such as the interrupt controller asynchronously and non-deterministically deciding to set the interrupt-pending bit. This ensures that we correctly model how the CPU handles interrupts, down to the precise point within the execution of a single instruction at which it checks for pending interrupts.

Second, separate ownership of each register allows the xv6 proof to distribute each register to the subsystem responsible for managing it. For instance, the xv6 proof defines one predicate that controls virtual address translation; that predicate, and all of the proof machinery around it, owns the `satp` and `tlb` registers. The proof also defines another predicate in charge of interrupt handling; that predicate owns the `stvec` register, the interrupt-enable bit of the `sstatus` register (owned through ghost state that subdivides the `sstatus` register into individual ghost bits that can be owned by different predicates), and sufficient memory locations to execute the interrupt handler, as well as other resources. Finally, the proof also has a separate invariant that owns the cycle-counter/timer register, and is used to enable timer interrupts. Distributing ownership of these registers among different predicates and invariants avoids the need to explicitly consider them in the rest of the proof. For example, the rest of the xv6 proof largely ignores the fact that each instruction involves one or two virtual memory translations for instruction fetch, and perhaps more translations for data loads and stores. Similarly, most of the proofs are independent of the fact that an interrupt can occur at any time (when interrupts are enabled).

In addition to full ownership of resources, CSL has a notion of a resource being persistent, which means that any number of threads can hold it at the same time. This is useful for representing read-only state that is accessed by multiple threads. For example, when the system is booting up, it initializes certain configuration registers, such as RISC-V's `medeleg` and `senvcfg` registers, and they are never modified again. Thus, after writing these registers, the

kernel proof transforms its exclusive ownership of the register into a persistent fact. This simplifies reasoning about these registers, since many parts of the kernel proof can now own this persistent fact stating the register's value.

Shared memory. To represent ownership of shared memory under TSO with load-load reordering, MachCSL provides an $a \mapsto_{\mathsf{TSO}} H$ resource, which states that the timestamped history of writes to address a is exactly H. MachCSL also provides a per-CPU resource representing the CPU's TSO timestamp, `timestamp_is` t, which can grow monotonically but never decrease. The combination of these two resources allows reasoning about the effects of reads and writes: a write extends the history, while a read returns some value from the history subject to the CPU's timestamp (for writes from other cores), or this core's own latest write.

Much of the xv6 kernel uses spinlocks to protect shared memory, which allows reasoning about memory as if it were sequentially consistent. To model this regime, MachCSL provides a points-to fact to represent ownership of a consistent physical memory location, $pa \mapsto_{\mathsf{Mem}} v$. Underneath, this is defined as $pa \mapsto_{\mathsf{TSO}} H$ together with an assertion that the local core's timestamp forces reads to observe the latest write.

The xv6 kernel runs with paging enabled, which means that instructions and data are accessed at virtual addresses, rather than physical addresses. The xv6 proof represents this using a $va \mapsto_{\mathsf{vMem}} v$ resource, which combines ownership of some (existentially quantified) physical memory address pa, $pa \mapsto_{\mathsf{Mem}} v$, together with a fact about the current virtual memory configuration, saying that virtual address va currently translates to physical address pa. This translation statement is represented using ownership of ghost state: the proof stores the current virtual memory mapping in a ghost variable, and each translation is a fact about a single element in this ghost map. The page-table predicate, mentioned above, connects the ownership of `satp` and `tlb`, as well as the physical pages comprising the 3-level page table, to the logical view established by that page table and TLB.

The xv6 kernel does not modify its own code. As a result, the xv6 proof makes all of the memory points-to facts for kernel code persistent. This allows freely sharing them between all proofs, so there does not need to be exclusive ownership of instruction memory. This also ensures that nothing in the kernel can modify the instructions themselves (since doing so would require the proof to obtain exclusive ownership of instruction memory).

Devices. RISC-V exposes devices by mapping their control registers in a separate part of the physical address space (MMIO). Reads and writes to these physical addresses do not work through memory points-to facts, because these registers do not behave like memory. MachCSL introduces explicit state for each device, in its device model, together with resources defining ownership of this device state. Device state is modified both by the device itself (e.g., a UART receiving and sending bytes in the background, or the disk controller executing DMA requests to read/write disk blocks) and in response to reads/writes to the device's memory-mapped control registers.

This device state in the xv6 proof is owned by an invariant, one for each device (UART, PLIC, and virtio disk in xv6, plus one for the external-interrupt pins that the PLIC drives into each HART). MachCSL requires the developer to prove that this invariant is maintained by the device's own operation, as well as by MMIO accesses from kernel driver code. To this end, the developer must prove that the logical device thread (representing the behavior of the device according to the MachCSL model) correctly executes against this invariant, in a style similar to what we described in §4 for kernel software itself. The proof for the kernel device driver follows exactly the same steps as any other kernel code that uses invariants. For example, in the case of the DMA-capable disk controller, the invariant associated with the disk owns both the device state and the physical memory used for the command queue, buffers of pending block reads/writes, etc. Ownership of command slots in the ring buffer is handed off from kernel to device and back, through the invariant, as a result of command submission and completion. Proving that the device upholds the invariant ensures that the kernel's driver code can safely reason about its interactions with the device through the abstraction of a CSL invariant over the device state, even in the presence of concurrent DMA operations to shared memory, interrupts, etc.

Load and store operations to device MMIO addresses follow a different specification from memory loads and stores. MachCSL proves separate specifications for every MMIO address (e.g., one specification for accessing the UART status register, another specification for storing into the UART TX FIFO, etc.). The specification uses $\Rrightarrow$ to open the device invariant and describe what effect that particular MMIO access has on the device state.

5.2 Execution

At the whole-instruction level, execution is captured in MachCSL by the wp `CpuLoop` resource, representing the fact that the HART can safely run `CpuLoop`, the infinite loop of RISC-V *cycles*; each cycle of the Sail model fetches and executes one instruction, or takes an interrupt or trap. The `CpuLoop` operation is indexed by a HART ID, which determines the set of registers that are used in executing each cycle; much like register points-to facts, we usually do

not explicitly show this HART ID index of CpuLoop, although it is important when reasoning about preemption of kernel threads and the scheduler resuming their execution on a different HART (Figure 11 shows it).

To reason precisely about the sub-instruction-level execution of a RISC-V HART based on the Sail model, MachCSL defines wp CpuLoop as a statement about executing the individual steps of the Sail RISC-V HART model. Specifically, MachCSL has a lower-level notion of wp Sail(m) $\{\Phi\}$, where m is the functional description of how the Sail RISC-V HART model executes, represented using a monadic encoding in Rocq. Here, Φ is the explicit postcondition that will be true after m executes. One notable difference between wp CpuLoop and wp Sail(m) $\{\Phi\}$ is that the control flow is now made explicit, and there is also an explicit postcondition, in contrast to the continuation-based wp CpuLoop. The reason is that, in our RISC-V operational semantics, the steps of a single cycle are statically defined by Sail's model: m is a fixed monadic program that terminates, so Φ can describe its result. In contrast, which instruction the next cycle executes is determined by dynamic state, such as page tables, memory contents, interrupts, etc., which is why instruction-level specifications take a wp CpuLoop continuation instead. MachCSL defines CpuLoop $\triangleq$ forever $\{cycleModel\}$, where *cycleModel* is the entire definition of the Sail model for a single cycle of RISC-V execution. wp CpuLoop has no postcondition because the processor executes cycles indefinitely, until powered off; all of the facts at the end of each instruction's execution are represented using CSL resources and passed into the proof of the next instruction's execution.

MachCSL uses standard sequencing combinators to reason about the steps of the Sail model. In particular, proving wp Sail(a; b) $\{\Phi\}$ reduces to proving wp Sail(a) $\{$wp Sail(b) $\{\Phi\}\}$, meaning that, to prove that Φ holds after running a; b, it suffices to prove that, after running a, there is a wp Sail(b) $\{\Phi\}$ resource that allows running b to establish Φ. A similar combinator works for sequencing with return values (i.e., a could return some value that b depends on, and in that case, Φ is a function of that return value).

This encoding allows MachCSL to state individual specifications for sub-instruction stages of execution. For example, in the Sail model as extracted to Rocq, there is a separate function used to perform address translation, `translateAddr`. The xv6 proof establishes wp Sail(`translateAddr`) $\{\Phi\}$, with a Φ that specifies how address translation works (i.e., address translation through a 3-level page table produces a result consistent with the abstract-state view of that page table according to the xv6 kernel's page-table invariant). Similarly, MachCSL proves sub-instruction specifications for the execute stages of different instructions, such as ALU operations, bitwise operations, etc. MachCSL also proves specs for all other stages of the Sail model, including instruction fetch, decode, trap handling, computing the next PC, etc.

Each individual wp CpuLoop specification, such as the ones shown in Figure 8, is then proven on top of these lower-level sub-instruction stage specifications. For instance, the spec for `AMOSWAP` combines a `translateAddr` spec for doing a virtual memory lookup of the program counter, together with an instruction fetch from physical memory, decode, another `translateAddr` to look up the physical address for the virtual address being swapped, then some register operations, then an exclusive memory write (to ensure atomic read-modify-write semantics) to perform the actual swap, etc. This ensures that the instruction-level specs at the wp CpuLoop level are completely faithful to the Sail model at the sub-instruction level. This includes both fine-grained concurrency between HARTs and devices, at the individual memory event level, and fine-grained power failures, which can occur between any two sub-instruction steps.

Non-HART execution. As mentioned above, devices are also represented by a logical thread of execution in MachCSL's semantics, and the developer is responsible for proving a WP-style statement for the execution of the device hardware model against the developer's stated invariant over the device state. The device WP is similar to the wp CpuLoop and wp Sail(m) $\{\Phi\}$ specifications we have already described, except that the low-level operations represent the steps that a device can take. Finally, there is a global power-cycling thread, also with its own set of operations (power-on and power-off); the developer also proves a WP for this power-cycling thread, which requires the developer to prove that any preconditions of starting execution of the kernel code on each HART are satisfied when that HART is started by power-on.

5.3 Adequacy

CSL specifications can be complex and can have complex implications. For example, if a specification inadvertently requires ownership of the same object twice in its precondition, that precondition can never be satisfied by any caller under CSL's rules, and the specification is vacuously true. Similarly, if a CSL specification promises the same object twice in its postcondition, it is unsatisfiable. The presence of proof callbacks ($\Rrightarrow$) further complicates the meaning of a specification.

To ensure that MachCSL specifications are meaningful, MachCSL provides an adequacy theorem, which turns a CSL-style specification about WP execution of the kernel into a lower-level statement purely about the execution of the system model (i.e., Sail's model plus MachCSL's shared-memory and device models), with no separation logic, WPs, etc. The adequacy theorem states that, for any execution of the system according to the operational semantics of our system model, as described in §3, the state of the system at each step will satisfy the invariants established in the proof. For instance, one implication of this adequacy theorem is that the durable disk state will remain consistent at all times.

One important job of the adequacy theorem is to ensure that all invariants used in the CSL proofs can be instantiated. In the absence of an adequacy theorem, it is easy for developers to assume some invariant holds, and prove functions with respect to that invariant, while not realizing that this invariant cannot be instantiated, and in fact the existence of the invariant might subtly imply false. To rule this out, the adequacy theorem must explicitly initialize every invariant.

The adequacy theorem also connects the CSL specifications and proofs to concrete states over which the system operates. For example, the file system in xv6 is proven correct using an invariant that holds over the contents of the durable disk state. This invariant states that the block free bitmap does not mark as free any blocks that are currently in use by some file, that the inode link counts are consistent, etc. The adequacy theorem provides a more concrete statement, about the system starting with the literal contents of the bytes from `fs.img` on disk. To this end, the adequacy proof shows that these initial bytes produced by `mkfs` satisfy the file system's invariant, bootstrapping the inductive reasoning across arbitrarily many power cycles.

6 Proving xv6 in MachCSL

This section describes the abstractions and specifications used in the MachCSL-based proof of xv6, as well as particularly interesting aspects of the proof itself.

6.1 Interrupt handling

The xv6 kernel enables interrupts during parts of its supervisor-mode execution. In particular, the kernel enables interrupts while executing system calls, so that long-running system calls can be preempted by a timer interrupt, and so that device interrupts can be serviced while executing long-running system calls. Similarly, the kernel scheduler on each core (HART) periodically enables interrupts while scanning for runnable processes, to check for timer interrupts or device interrupts that might be needed to wake up a process that might be waiting for UART input or disk I/O.

Interrupts introduce a tricky source of concurrency, because when the kernel is executing with interrupts enabled, it can be preempted by an interrupt at any time. The interrupt handler executes on the same kernel stack as the interrupted thread. Furthermore, the interrupt handler could put the thread to sleep (e.g., calling `yield` in the timer interrupt to schedule another process), and the same process could be resumed on a different HART instead.

The xv6 proof addresses these challenges by introducing a resource responsible for handling interrupts and CPU migration. In particular, the xv6 proof defines a resource sie_cap_gpr, which exposes several arguments. One of these arguments is b, a boolean flag indicating whether interrupts are currently enabled. Another argument is n, the number of free stack locations on the current stack. Another argument is m, a map giving the values of the general-purpose registers on the current HART.

Every developer-visible instruction specification in MachCSL takes sie_cap_gpr as a precondition and returns it to the continuation. This allows the proof of every instruction to do an inductive loop over an arbitrary number of interrupts that could arrive at that instruction boundary (corresponding to wp Sail(`dispatchInterrupt`) {Φ}, where `dispatchInterrupt` is the Sail functional model of checking for pending interrupts and jumping to the trap handler).

Register migration. Instruction specifications use the m argument of sie_cap_gpr to refer to the values of general-purpose registers, rather than the raw $r \mapsto_{\mathsf{Reg}} v$ resource shown earlier. The reason is that the raw register points-to resource is tied to that register on a specific HART ID. If the proof of some function owned this resource, but was preempted by an interrupt and migrated to another HART, it would not be able to meaningfully use that resource on the new HART ID, because the ID of the HART on which the code is executing might differ from the ID of the register points-to fact. When the interrupt handler resumes a thread on a different HART, it ensures that all of the saved registers are restored, and therefore the preempted thread resumes execution with the same exact m.

Stack reservation. The number of available stack locations n takes into account the total stack size, and the number of stack locations that need to be available to execute the trap handler. When the kernel executes an instruction to disable interrupts (writing to the `sstatus` register), the b argument of sie_cap_gpr flips to false, and the number of

available stack locations increases by the trap handler's reserve amount. When the kernel re-enables interrupts, the precondition of the instruction spec that writes this bit to `sstatus` requires the caller to prove that the current n stack budget is large enough to accommodate the trap handler's stack use, and subtracts that stack use from n (reserving it for the trap handler).

Trap handler address. The sie_cap_gpr resource owns the `stvec` register, and existentially quantifies over the trap handler's address. In particular, sie_cap_gpr states $\exists\, h.\ \texttt{stvec} \mapsto_{\text{Reg}} h * (\texttt{pc} \mapsto_{\text{Reg}} h \mathrel{-\!\!*} \ldots \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{CpuLoop})$. This says that `stvec` has some address h such that, if the PC were to jump to h, along with some other conditions, it would be safe to keep executing.

When interrupts are disabled, sie_cap_gpr gives up ownership of `stvec`, allowing the rest of the kernel code to manipulate `stvec`. This is important when the kernel jumps to executing user-mode code. In xv6, there is a different trap handler for user-mode traps, which, among other things, implements demand allocation of zero pages for process memory through page faults, and executes system calls in response to an `ECALL` instruction. Once the kernel disables interrupts, the proof gets back ownership of `stvec`, allowing it to install a different handler address by writing to that register, and use that to establish a different interrupt-safety invariant that holds while executing instructions in user mode.

6.2 Page table and TLB

To factor out virtual memory translation from the rest of the kernel proof, the xv6 proof introduces a resource that owns the page-table machinery. In particular, strans_inv owns the `satp` register, which points to the root of the page table, as well as the physical pages comprising the 3-level page-table structure, and the Sail model's `tlb` state. The proof also maintains a map in ghost state that represents the logical mapping implemented by the kernel's page table. The strans_inv resource states that the TLB must be consistent with the mappings in the page table.

A/D writeback. One complication is that the page-table hardware can write back accessed (A) and dirty (D) bits to page-table entries, if those pages were in fact accessed or modified by some HART. At the same time, the kernel uses a single page table shared across all HARTs, so the A/D writeback on one HART can modify the page table that another HART is walking. To prove the correctness of this, the xv6 proof places ownership of the physical page-table pages into an invariant. For every instruction, the execution of the page-walk and TLB phases of the Sail model must be able to open the invariant and modify the physical page table in memory, updating its A and D bits. To allow this, strans_inv existentially quantifies over A and D bits for every page, allowing arbitrary A/D bit updates.

Translation tiers. In xv6, the kernel can operate in three different address-translation regimes. When the system first boots up, every HART is running in machine (M) mode, which by definition does no address translation. The xv6 initialization code then switches to supervisor (S) mode, but running without a page table ("Bare" mode in RISC-V terminology), which appears as an identity mapping between virtual and physical addresses. Later, xv6 allocates a page table, containing both identity and non-identity mappings (a kernel stack for every kernel thread, and a trampoline page containing code that switches between kernel and user page tables, at the top of the virtual address space). Once the kernel installs this page table, it can start accessing non-identity virtual addresses, such as per-kernel-thread stacks, and the trampoline to switch to user mode.

One complication of this bootup sequence is that, while one HART may have already initialized and switched to using the kernel page table (where non-identity mappings are present), other HARTs might still be executing early in the boot sequence, still in the identity-mapping regime without the non-identity mappings. As a result, a kernel virtual memory points-to fact ($va \mapsto_{\text{vMem}} v$) that is valid for accessing memory on one HART is not necessarily valid for accessing memory on another HART.

To define sound rules for accessing memory through a $va \mapsto_{\text{vMem}} v$ resource, the xv6 proof introduces the notion of a *kernel translation tier*, which takes advantage of the fact that kernel mappings grow monotonically in xv6. Tier 0 represents identity mappings that are valid in all three regimes (M mode, Bare mode, and with the kernel page table). Tier 1 represents mappings that are present in the kernel page table. Tier 0 mappings are a subset of tier 1 mappings. Every kernel virtual memory points-to fact is indexed by a tier (0 or 1), even though we have not explicitly shown this tier index in this paper's presentation. The state of the current HART also keeps track of the tier that this HART has reached; this is an argument kt of sie_cap_gpr. The specifications for memory load and store instructions require that the tier of the memory points-to fact be less than or equal to the tier of sie_cap_gpr.

Switching user and kernel page tables. When xv6 switches to running user code, it installs the user process page table, which does not map the kernel's code or data. The user page table contains the user's own memory, plus two special pages: a trampoline page, containing kernel code to jump in and out of the kernel (which is responsible for switching page tables), and a trapframe page, which is used to save the user's registers when trapping into the kernel.

To switch between user and kernel page tables, the trampoline assembly code executes a conservative but safe sequence: `sfence.vma` to flush the TLB, then load the `satp` register with the new page-table root, then `sfence.vma` to flush the TLB again. One complication is that, while the HART is executing between the two TLB flushes, the TLB might contain mappings from *either* of the two page tables. To pin down the precise reasoning for why this sequence is safe, the xv6 proof states that, in this page-table-switching window, the TLB must contain only the mappings that are present in both the user and kernel page tables (namely, the trampoline code page).

6.3 Context switching

The kernel uses `swtch` to implement context switching between kernel threads. In particular, when a kernel thread decides to `yield` or go to sleep, it uses `swtch` to switch to executing a per-HART scheduler thread. The scheduler thread, in turn, loops over all processes, and when it finds a `RUNNABLE` process, it uses `swtch` to switch to executing that kernel thread. Timer interrupts can preempt execution of a process, either in user mode or in kernel mode, and force that process to give up the CPU by calling `yield`. The function signature of `swtch` is shown in Figure 10 in C syntax, although the actual implementation of `swtch` in xv6 is in assembly.

```
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  ...
  uint64 s11;
};

void swtch(struct context *old, struct context *new);
```

Figure 10: Signature of the `swtch` context-switching function.

The specification of `swtch` needs to capture the fact that a scheduler can resume executing a kernel thread by using `swtch`, regardless of what the kernel thread was doing at the time it was preempted or decided to sleep on its own. The xv6 proof captures this notion in the specification of `swtch` by storing a wp CpuLoop inside the resource that defines a valid `struct context`. own_context *ptr c* says that the `struct context` at address *ptr* stores the register values represented by tuple *c*. valid_context *ptr* says that there is some tuple *c* of registers saved at *ptr*, and, if any HART *h* restores the callee-saved registers (`sp` and `s0`–`s11`) from *c*, that HART can resume execution at *c.RA*. The HART ID *h* is universally quantified, because the HART that resumes the kernel thread need not be the one that saved its context; the resulting wp (CpuLoop *h*) states that it is HART *h* that continues executing the thread.

The spec for `swtch` itself (Figure 11) says that, as long as the caller can prove that it owns the memory locations for storing the old context (first argument, passed in `a0`), and there is a valid context at the new pointer (second argument, passed in `a1`), and the caller can prove that it is safe to resume execution at the return address with the callee-saved registers restored, then the caller can provably invoke `swtch`. The spec makes the HART ID argument of CpuLoop explicit (§5 otherwise omits it): `swtch` is invoked on HART *h*, but the continuation must hold for any HART h', since `swtch` might return on another HART after the scheduler there resumes the thread. Not shown in this spec is that, if the caller knows the scheduler will never resume this process again (e.g., a killed process exiting), the caller's proof need not supply this continuation WP.

6.4 Locks

The xv6 kernel uses spinlocks to coordinate access to shared data structures between concurrent threads. §4 showed a simplified version of the xv6 spinlock, along with basic CSL-style specifications for the spinlock. This subsection describes several complications of how xv6 implements and uses spinlocks, and how the proof is designed to address them.

own_context $ptr\ c \triangleq$
  $ptr + 0 \mapsto_{\text{vMem}} c.RA *$
  $ptr + 8 \mapsto_{\text{vMem}} c.SP *$
  $ptr + 16 \mapsto_{\text{vMem}} c.S0 *$
  ...
  $ptr + 104 \mapsto_{\text{vMem}} c.S11$

valid_context $ptr \triangleq$
  $\exists\ c.$ own_context $ptr\ c *$
  $\forall\ h\ m.$ sie_cap_gpr m ... $-\!\!*$
    callee_saved m = (c.SP, c.S0, ... , c.S11) $-\!\!*$
    pc $\mapsto_{\text{Reg}}$ c.RA $-\!\!*$
    wp (CpuLoop h)

kernel_text $-\!\!*$
pc $\mapsto_{\text{Reg}}$ swtch $-\!\!*$
sie_cap_gpr m ... $-\!\!*$
($\exists\ c.$ own_context m[a0] c) $-\!\!*$
valid_context m[a1] $-\!\!*$
($\forall\ h'\ m'.$ pc $\mapsto_{\text{Reg}}$ m[ra] $-\!\!*$
    sie_cap_gpr m' ... $-\!\!*$
    callee_saved m = callee_saved m' $-\!\!*$
    ($\exists\ c.$ own_context m[a0] c) $-\!\!*$
    valid_context m[a1] $-\!\!*$
    wp (CpuLoop h')) $-\!\!*$
wp (CpuLoop h)

Figure 11: Simplified specification for swtch.

Disabling interrupts for spinlocks. xv6 couples spinlocks with interrupt handling, because it is important that a CPU holding a spinlock not be preempted by an interrupt; otherwise, the kernel could deadlock. In the spinlock implementation, this is done using two functions, push_off and pop_off. push_off increments a per-CPU interrupt-disable counter, and pop_off decrements it. push_off saves the original interrupt-enable bit, so that when pop_off brings the counter back to zero, it correctly restores the state (either enabled or disabled).

This shows up in the specifications through two arguments to sie_cap_gpr: the current depth d of push_off calls, and the original interrupt-enable bit *eb* when the first push_off ran. When d is non-zero, the interrupt-enable bit b is false. When d goes from 1 to 0, b becomes *eb*. When d goes from 0 to 1, *eb* becomes b. When there is no active push_off on the current CPU (i.e., $d = 0$), the kernel can explicitly toggle the interrupt-enable bit by modifying the sstatus register; this toggles the b argument of sie_cap_gpr.

One side effect of this design is that spinlocks must be released on the same CPU on which they are acquired. This is represented by the acquire function returning an additional lock_held resource that is tied to the specific CPU on which the acquire executed. This might not be the CPU on which acquire started, because acquire might have been preempted before it disabled interrupts, but it is the CPU on which acquire returned. release requires passing this lock_held resource for the current CPU.

Avoiding deadlocks. The spinlock implementation in xv6 checks whether the caller is already holding the spinlock it is trying to acquire, and if so, panics. Moreover, xv6 tries to avoid deadlock through spinlocks by maintaining a consistent locking order, so that locks are never acquired in conflicting order by different CPUs. The spinlock specification addresses both of these problems by maintaining the current lock-held set for each CPU, as the *lkset* argument of sie_cap_gpr. The precondition of acquire requires the caller to prove that the lock it is about to acquire is ranked higher than any other lock in the current lock-held set. This ensures both that the lock it is about to acquire cannot already be held by the same CPU (preventing the panic) and that locks are never acquired in a conflicting order (preventing deadlock).

To be precise, the MachCSL top-level adequacy theorem currently does not guarantee the absence of deadlock (since that is a liveness property), but the xv6 specification for acquire forces the rest of the kernel code to follow the lock-ordering discipline, and in practice eliminates the possibility of spinlock-based deadlock.

Dynamic allocation. In xv6, pipes are implemented by dynamically allocating a page of kernel memory to store the state of a struct pipe. The implementation of pipes allows concurrent reads and writes from multiple cores, and uses a spinlock to protect the pipe state. This spinlock itself also lives in the dynamically allocated page of memory. This complicates the spinlock specification because the spinlock's invariant is no longer true at all times: at some point, the page of memory will be deallocated, when the pipe is fully closed, and from that point on, the page might be used for something else, which is inconsistent with the pipe's locking discipline. The xv6 proof solves this by making the lock invariant *cancellable*. Namely, in order to acquire the lock, the caller must prove that the pipe's reference count has not reached zero. This is established for each caller that reads from or writes to a pipe by the fact that the struct file referring to either end of the pipe holds a reference on the struct pipe. When the reference count reaches zero, the proof can conclude that no future caller can ever satisfy the precondition to acquire the lock, and it is safe to cancel the pipe's lock invariant, which allows reclaiming full ownership of the pipe's memory page, which in turn allows freeing it.

Sleeplocks. In addition to spinlocks, xv6 implements sleeplocks for cases when the thread holding a lock may need to go to sleep (i.e., yield the CPU to other processes), which shows up in the file system when it needs to hold a sleeplock while waiting for disk I/O. For example, the file system locks an inode, to ensure other kernel threads do not access or modify the state of that inode, while reading the inode in from disk or writing it back.

One requirement for sleeplocks is that the caller must not hold any spinlock while trying to acquire a sleeplock. Otherwise, the sleeplock acquire might switch to the scheduler while holding a spinlock, which could result in deadlock. To ensure this does not happen, the specification for the sleeplock acquire requires that the caller have an empty lock-held set, as specified by the sie_cap_gpr *lkset* argument.

An exception to this rule shows up in the file system code for releasing the last reference on an inode. When the caller drops the last reference to an inode, and it observes that the inode now has a zero link count, it needs to then free the inode on disk. The xv6 kernel implementation acquires the sleeplock while holding the spinlock protecting the inode's in-memory reference count. The reason this is safe is that there cannot be any other concurrent threads trying to acquire this sleeplock, since the caller holds the only in-memory reference to the inode (the inode's refcount is 1). To prove this is safe, the sleeplock specification introduces a resource to represent any active sleeplock holder. Trying to acquire a sleeplock requires supplying such a resource (which, in practice, is a share of an in-memory reference to the inode). Any acquired sleeplock holds this resource until the sleeplock is released. When the caller knows that it holds the only reference, it can prove that there cannot be any such resource held by the sleeplock, and therefore, acquiring the sleeplock will succeed immediately, and will not go to sleep.

6.5 Memory allocation

xv6 implements a page allocator: `kalloc` returns a fresh page of memory, and `kfree` puts it back on the free list. The essence of the specification for `kalloc` and `kfree` is a standard CSL encoding of memory allocation: `kalloc` returns ownership of 4096 bytes of memory, at the returned address, and `kfree` requires the caller to give up ownership of all 4096 bytes. `kalloc` can non-deterministically fail if there is no memory available, in which case it returns 0.

One complication is that xv6 uses `kalloc` during bootup to initialize system data structures like the page table, the per-process stacks, etc. This initialization does not check for `kalloc` returning 0 because it will always succeed, since no other processes are running and there must be memory still available. However, the standard `kalloc` spec does not allow the caller to conclude that allocation will succeed, because it is aimed at running in a concurrent setting.

The xv6 proof addresses this by having a single spec cover both modes of operation: `kalloc` starts out operating in a *counted* regime, represented by a kalloc_env (Some n) resource, which is *exclusive*: only one CPU can own it at a time. The n argument represents the number of free pages available for allocation. The spec for `kalloc` says that, if the caller holds kalloc_env (Some n), then allocation must succeed if $n > 0$. This allows the bootup sequence to prove that all allocations will succeed, by precisely tracking the number of free pages. Once CPU 0 finishes the boot-time allocations, the proof exchanges kalloc_env (Some n) for kalloc_env None, which represents the *concurrent* regime of `kalloc`. In this regime, allocation can non-deterministically return 0. The benefit of this regime, however, is that kalloc_env None is persistent, which allows all cores to own it and allocate memory concurrently. Once the allocator enters the concurrent regime, it cannot go back to the counted regime.

The traditional specification for a memory allocator requires the caller to give up ownership of $a \mapsto_{\mathsf{Mem}} v$ for every memory location that they are freeing. This is subtly too strong in a TSO shared-memory model, because $a \mapsto_{\mathsf{Mem}} v$ implicitly says that address a has a stable value if read on the current CPU. This is not strictly necessary to free a page of memory: the allocator does not care if the current CPU sees a stable value for the memory being freed. We had to change our `kfree` specification to accept a weaker precondition of owning every address being freed, without requiring it to be stable.

6.6 File descriptors

xv6 uses reference counts in `struct file` to keep track of how many outstanding references exist to a particular open file. This reference count is incremented every time a user process calls `dup` or `fork`, and decremented when a file descriptor is closed. The kernel implementation does not do overflow checking when incrementing the reference count. The reason this is safe is that there can be only a bounded number of open file descriptors that can hold such a reference. To formally prove this argument, the xv6 proof introduces a notion of a file reference slot, which is a token representing the right to increment a file's refcount. There is a fixed number of file reference slots in each era (the period between one power-on and the next), distributed at boot time across all `struct proc` entries, each of which receives one slot per entry in its file descriptor table (`NOFILE` pointers to `struct file`). When a process opens a file descriptor, the proof transfers ownership of the slot resource to the `struct file`; when the file descriptor is closed, the proof moves the unused reference slot back into the process invariant. Each `struct file` keeps a list of these

slots, and maintains an invariant that its refcount is equal to the number of slots in its list. Since only a bounded number of slots are distributed at system boot time, this allows the proof to conclude that the refcount increment cannot overflow.

A second complication with file descriptors is that multiple processes might have a file descriptor referring to the same `struct file`, yet the `struct file` has a single reference on an underlying `struct pipe` or inode. As a result, when processes concurrently read from file descriptors that actually represent the same underlying `struct file`, they must share the fact that they are holding a reference on the `struct pipe` or inode between them. For example, recall that acquiring a spinlock on `struct pipe` requires proving that the pipe's refcount is non-zero, as part of the pipe's cancellable spinlock. This is represented using fractional ownership of a resource representing a unit refcount on either a `struct pipe` or inode.

6.7 File system

xv6 has a traditional Unix file system with directories and files, and system calls for creating, reading/writing, and removing them. The file system—without the disk driver, pipes, and file descriptor system calls—constitutes ~30% of the xv6 kernel in terms of lines of code. Formally reasoning about the file system code is tricky because the file system handles crash recovery (by executing each file system operation as a transaction), has substantial internal concurrency (system calls involving different inodes may run in parallel on different CPUs, and commit as a group), and must handle tricky Unix semantics (e.g., a file can be unlinked from a directory but still be in use by another process that has the file open). This subsection highlights a few interesting issues in the xv6 proof of the file system.

File system durability. A key property of the file system is that it must be consistent across crashes. This is formulated in the xv6 proof using an invariant about the durable on-disk state; this invariant persists across reboots. xv6 uses a write-ahead log to update the disk contents in a way that is atomic with respect to crashes. The xv6 proof represents this by introducing two views of the disk contents: D, which represents the durable state on disk, and M, which represents the in-memory state that will be written to the log and committed.

The D state reflects what the on-disk state will be after recovery finishes (i.e., after applying the log). This means that D is updated to match M when the log header recording the commit is written to the disk. The `recover_from_log` function installs the log after a crash and reboot; its specification says that it keeps D unchanged.

The M state reflects the state of the file system present in the buffer cache. For instance, when a system call executes, it updates blocks containing the relevant file data, inodes, bitmaps, etc., and hands the blocks to the write-ahead log for eventual commit. The M state is updated at the time blocks are handed to the write-ahead log layer. When the log commits transactions, it updates D to be equal to M. This allows the log to prove that the on-disk state, which is now $D = M$, satisfies the file system consistency invariant, because the in-memory state M satisfied the consistency invariant before commit.

One complication with the in-memory M state is that it is not always a consistent state of the file system. For example, during the execution of `sys_unlink`, the kernel first updates the parent directory, removing the directory entry, and then updates the child, dropping its link count. Between these two operations, the M state is not consistent. Furthermore, ownership of different parts of M is distributed among many locks, because the implementation is concurrent: different CPUs may be running system calls that have locked different parts of the file system.

To prove that the file system is consistent at commit time, the xv6 proof maintains an invariant that every write-locked inode (i.e., an inode locked by a system call that may modify it) must be part of a transaction. At commit time, the write-ahead log waits until all of the in-flight transactions have finished before writing the set of all modified blocks to the log. This ensures that, at commit time, no inode can be write-locked, because there are no pending transactions. This allows the commit proof to observe that all inodes are in a consistent state, because all of them satisfy the inode's lock invariant.

Writing the log header. The xv6 kernel commits a transaction by writing the *log header* block, which lists the blocks in the log, to disk. If xv6 experiences a crash, xv6 will check the log header on reboot. If there is a committed transaction in the log, it will install the blocks modified by the transaction into their home locations on disk. The implementation of xv6 ensures commits are atomic by writing a single block and waiting until the disk acknowledges that the write has completed.

There are two complications that the proof must handle: (1) an xv6 disk block is two disk sectors, and only single-sector writes are atomic; (2) unless explicitly disabled, the virtio disk uses a writeback cache and writes sectors asynchronously: if the write operation completes, the disk guarantees that the sector is in its writeback cache but it may not be on the disk itself, and thus a power failure may cause the acknowledged sector writes to be lost. The

reason that commits are synchronous and atomic in xv6 is that (1) xv6's log header fits in a single sector, and (2) the xv6 disk driver does not negotiate `VIRTIO_BLK_F_FLUSH`, making disk operations write-through.

The proof establishes write-through behavior as a property of the xv6 driver. The proof of `virtio_disk_init` computes the feature bits that the driver writes to the device, and the device protocol invariant records that `VIRTIO_BLK_F_FLUSH` was not negotiated, from which the proof concludes that the cache is empty whenever a request completes.

Reasoning about what is on disk after a crash requires setting up ownership carefully, because a crash destroys all threads and every resource they own; the persistent state of the disk is therefore owned by a crash invariant that survives power cycles. A disk write is specified through a *write permit*: a proof callback (a $\Rrightarrow$ resource, §4) over the crash invariant that the proof constructs when the driver issues the request and that the proof applies at the moment a sector lands on disk. Since a block write consists of two sector writes that may be written in either order to the disk, and both need to update the log's ghost state describing the disk, the proof cannot hand out two independent permits, one per sector. Instead, a request carries a single permit for a set of sectors; the device picks which sector to write first, and applying the permit to that sector returns the permit for the remaining sectors, threading the ghost state through from one sector write to the next. The final permit, when no sectors remain, delivers the postcondition that the caller receives when the write completes.

Atomicity of the commit follows from the layout of the xv6 log header, which fits in the first 124 bytes of its block, and from the fact that recovery reads only those bytes. The proof builds the header write's permit for both sectors. When the disk writes the block's first sector (which holds the 124-byte header), the proof switches the log header from the old transaction to the new one in one step; writing the other sector, whether it happens before or after the first, changes nothing that recovery reads. Together with write-through completion, this shows that when the write of the log header (`write_head`, called by `commit` from `end_op`) completes, the new log header is on disk, and that a crash in the middle of the write leaves either the old or the new log header, but never a mix.

Log budget for transactions. The xv6 log is bounded in size, and the code is carefully written to ensure that any transaction writes no more than `MAXOPBLOCKS` blocks, which in turn ensures that every transaction will be able to commit.

To prove that each transaction stays within its budget of log blocks, we originally used a worst-case bound by counting the number of times the kernel logs a write in a transaction, and requiring that this never exceed `MAXOPBLOCKS`. This turned out to be insufficient because the kernel relies on log absorption to stay below this bound. In particular, the kernel might update the free bitmap block multiple times within a single transaction (e.g., freeing multiple disk blocks when deleting a file), or update the same data block multiple times (e.g., writing the directory entries for `.` and `..`); those writes are absorbed by the log and do not count separately towards the transaction's budget.

We extended the spec to also track the set of blocks logged by the transaction so far; a `log_write` of a block already in that set is absorbed and does not consume one of the transaction's `MAXOPBLOCKS` units. The function `begin_op` that starts a transaction for a system call creates a new resource for this set and passes it down to functions that call `log_write` to update the set in the proof. `end_op`, which commits a transaction, requires that the set be no larger than `MAXOPBLOCKS`.

Group commit for `namei`. Handling absorption within a transaction was not enough to reason about the budget of log blocks in xv6. Consider a process that calls `namei("/a/..")`. The kernel runs `namei` inside a transaction because when `namei` is done with an inode (e.g., the inode for "/a"), it calls `iput`, and `iput` may discover that `namei` has the last reference to the inode because concurrently another process unlinked "/a", and thus `namei`'s `iput` logs the effects of the `unlink`.

Now consider a process calling `namei("/a/../b/../c/../d/..")` with a concurrent process unlinking each directory. If a racing `unlink` happens for every path element ("/a", "/b", "/c", and "/d"), `namei`'s transaction runs the risk of growing to an arbitrary number of blocks. Each time it calls `iput` for the inode of a path element, `iput` will observe a zero link count and log the `unlink`.

The reason that `namei`'s transaction stays within the `MAXOPBLOCKS` budget is that if a concurrent `unlink` races with `namei`, then the `unlink` must have also already written that exact inode into the log (when it did the `iupdate` to drop the inode's `nlink`), so the `iupdate` in `namei`'s `iput` gets absorbed with that other transaction. So `namei` stays within its budget because of cross-transaction absorption. We asked the agent to extend the proof to do cross-transaction accounting of log blocks, and that allowed the agent to complete the proof for `namei`.

Buffer cache. In xv6, allocating a buffer in the cache to hold a disk block for a block number is a distinct operation from loading the bytes of the block's content from disk. To read a disk block, the kernel calls `bread`, which calls `bget` to check the buffer cache. If a block is not present in the buffer cache, `bget` allocates an entry in the buffer cache under the `bcache.lock` lock, sets `b->blockno` to the block number of the block, increases `b->refcnt` from 0 to 1, sets `b->valid` to false, and releases the `bcache.lock`. Then, `bget` acquires the sleeplock `b->lock` and returns `b` to `bread`, which reads the data from disk and marks the buffer valid. When the kernel is done with the buffer `b`, it calls `brelse`, which decreases `b->refcnt`.

A complication for the proof is that between releasing `bcache.lock` and acquiring `b->lock` in `bget`, the kernel holds no lock, and the proof must argue that the buffer was not reused for a different block number, based on the fact that the kernel does not reuse a buffer with `b->refcnt` ≥ 1, even if `b->valid` is false. Another complication is that the proof must argue that a later `bget` for a block number in the buffer cache returns the current content of the block (which may differ from the on-disk contents while the block sits in the write-ahead log).

The proof addresses both complications by giving each buffer its own invariant that owns the buffer's contents (`b->valid`, `b->data`, and a share of `b->dev` and `b->blockno`) together with a ghost resource describing the block's current logical content, rather than associating the contents with either lock. The sleeplock `b->lock` protects only a checkout token: `bget` exchanges this token for the buffer's contents when it acquires the sleeplock, and `brelse` puts the contents back into the invariant in exchange for the token before it calls `releasesleep`, since a waiting `bget` may return as soon as the sleeplock is free, before `brelse` has decremented `b->refcnt`.

The proof tracks reference counts using ghost state under `bcache.lock` and fractional ownership: every outstanding reference owns a fragment of a per-buffer counter, and the invariant ties the counter to `b->refcnt`. The recycling path of `bget` needs the counter to be zero in order to rewrite `b->blockno`, so any thread holding a reference fragment—including one that has released `bcache.lock` but not yet acquired `b->lock`—can conclude that its buffer still holds its block number, regardless of `b->valid`.

For a buffer with `b->valid` false, the invariant holds the ghost resource for the block's content, which the thread that performs the disk read (whichever thread first acquires the sleeplock) uses to establish that the bytes it read match the block's logical content; for a valid buffer, the invariant states that `b->data` agrees with that content, which is what a later `bget` relies on. The proof of the buffer cache never interprets the content resource itself: it is a parameter supplied by the logging layer, which instantiates it with its own ghost state tracking whether the block's current content is on disk or in the write-ahead log.

Link consistency without global statements. The consistency of the file system, both on disk and in memory, requires that the directory structure be consistent with the link counts. At some level, this is a global property: the link count of each inode depends on the directory entries present in the rest of the file system. However, for performance, the xv6 implementation uses fine-grained per-inode locks when it runs. To avoid reasoning about the global file system state each time the on-disk or in-memory state is modified, we introduced a ghost resource to keep track of inode references in a modular way.

Specifically, each inode owns a ghost variable tracking its outstanding link count, and the inode's own invariant says that its `ip->nlink` field is equal to the number of outstanding links according to the authoritative ghost state. Every directory that has a link to an inode maintains ownership of a unit fragment of this ghost state, representing the fact that it incremented that inode's `nlink` by 1. This resource allows `sys_unlink` to justify decrementing an inode's `nlink` when it is removed from a parent directory, because the parent directory can return this unit fragment of ownership, proving that the `nlink` will not underflow when decremented.

A further invariant that the file system must maintain is that the `..` entry in every directory correctly refers to its parent inode. The reason is that, when a subdirectory is deleted, the implementation of `sys_unlink` in xv6 (which subsumes `rmdir`) decrements the parent directory's link count. The proof must establish that the child directory indeed held a reference on the parent directory. To do this without explicit global consistency properties, the xv6 proof includes the parent directory's inode number in the link-counting ghost state, for inodes that happen to be directories. (The parent directory's inode for files is not well-defined, because xv6 supports hard links.) This part of the ghost state allows each directory's invariant to say that (1) the inode number of the `..` entry is the parent directory's inode number from the ghost state, and (2) any child directory refers back to the correct parent directory's inode number.

The key benefit of this design is that it allows the xv6 proof to state file system consistency properties, such as link-count consistency and `..` consistency, in a local manner, specified in the independent lock invariant of each inode. This reduces the amount of global reasoning needed when performing local operations.

iget. The `iget` function takes an inode number, `inum`, as argument and returns the inode associated with it. The naïve spec for `iget` states that it returns an allocated inode. That spec is not strong enough, however, for concurrent system calls: another thread may have freed the inode and reused the `inum` for another inode. The naïve spec does not forbid it: the inode returned is allocated but is not the inode the caller intended to look up, which makes it difficult for callers of `iget` to reason about their correctness. Instead, `iget` has a stronger premise: it requires that the caller provide a resource that pins down the identity of the inode named by `inum` and rules out its being freed.

Callers of `iget` fulfill the premise in different ways. For example, `dirlookup` in `namex` fulfills it with evidence that it holds the parent directory locked and that the directory is still linked (i.e., `nlink` > 0). `ireclaim`, which scans the disk at boot for orphaned inodes and frees them, fulfills the premise with an exclusive *boot token*, which exists only during boot and demonstrates that `ireclaim` runs while it owns all inodes and xv6 runs sequentially. `ialloc` fulfills it with exclusive ownership of the freshly allocated inode, which, as we discuss below, is the hardest instance.

The premise relies on an invariant that entries exist only in linked directories (an unlinked directory is empty except for `.` and `..`); to preserve it, system calls that create names (e.g., `create` and `sys_link`) must check that the directory they write to is still linked.

The xv6 kernel had none of the necessary `nlink` checks: `namex` would follow `..` in an unlinked directory, and `create` and `sys_link` would write names into one. The proof got stuck trying to fulfill `iget`'s premise in `namex`; the stuck proof turned out to reflect a real bug with several manifestations, including a kernel panic (§11).

ialloc. When allocating an inode, the xv6 file system calls `ialloc` to get an inode number that is not in use. `ialloc` scans the inode blocks to find an unused inode number, marks the inode on disk as allocated by calling `log_write`, and calls `brelse` to release the block holding the inode. Then, it calls `iget` on the inode number to get a `struct inode` representing that inode. `ialloc` does not hold any locks between calling `brelse` and `iget`, and thus the proof must establish that no other thread could have modified the inode (e.g., freed it); that is, `ialloc` must discharge `iget`'s premise for an `inum` that is in no directory at all.

Initially, the agent could not prove this claim, and on one attempt changed the xv6 source to not do the `brelse` to make the proof go through, which we caught because the agent could not push to the official xv6 repo. On another attempt the agent introduced a new axiom `create_fresh_ty`. Our CI report of the axioms that the top-level theorem depends on caught this attempt (§10). In the end, it turned out that `ialloc` was not provable because `iput` had a bug (§11).

Proving `ialloc` after the `iput` fix turned out to still be difficult: the proof must make the argument that the allocated `inum` is not in the file system tree and thus no thread could have found the `inum` and modified the corresponding inode. This argument is further complicated because an inode might have been unlinked (and not be in the file system tree) while a file descriptor is still pointing to that inode; the agent had to argue that `iput` could not have freed such an inode yet because the file descriptor still holds an outstanding reference to it. Finally, `ialloc` must also rule out `ireclaim` as a concurrent writer, which it does using the same boot token that `ireclaim`'s own premise rests on: once boot finishes, the proof irreversibly exchanges the boot token for a persistent witness that boot is over, and `ialloc` requires this witness.

This proof relies on the link-count ghost state, together with a companion ghost state recording each inode's type (zero for a free inode), which lets the proof conclude that any inode with an outstanding link is allocated.

iput. The `iput` function puts an inode `ip` back into xv6's inode cache: it decreases `ip->ref`, so that the `struct inode` can be recycled for another inode once the count reaches zero. If `ip->nlink` is also 0, it frees the inode too. To read `ip->ref`, `iput` holds `itable.lock`, and if `ip->ref` is 1, it acquires the `ip->lock` sleeplock to free the inode. The implementation of `acquiresleep`, in turn, acquires the process's `p->lock` if another thread is holding the sleeplock. However, `kfork` holds `p->lock` across `idup`, which takes `itable.lock`. In this scenario there is no global ordering of locks.

The proof of `iput` must establish that this scenario cannot happen, which is true because, if `ip->ref` is 1, no other thread could be holding `ip->lock`, and `acquiresleep` will not go to sleep. To handle this case, we have two specifications for `acquiresleep`: the general one, which may sleep and therefore requires an empty lock-held set, and a non-blocking one, which requires the caller to prove that no thread can currently hold the sleeplock and therefore allows any lock-held set.

To be able to use the second specification, the proof associates with each inode's sleeplock a ghost counter of potential holders. The counter has two parts: an authoritative total, which is stored in the lock invariant of `itable.lock` next to the inode's reference count, and fractions that add up to this total, which the proof hands out to the holders of references to the in-memory inode (a fraction per reference, since many `struct file`s may share one

inode reference and race for its sleeplock). Acquiring the sleeplock deposits the caller's fraction in the sleeplock's invariant, and releasing it takes the fraction back; a thread can thus hold or wait for the sleeplock only if it owns a fraction.

When `iput` observes `ip->ref` equal to 1 while holding `itable.lock`, the ghost state for the reference count tells the proof that the reference `iput` owns is the only one, and therefore that `iput`'s fraction is the entire total recorded in the lock invariant of `itable.lock`. `iput` returns its fraction to the lock invariant, which brings the total to zero. A total of zero means that no thread owns a fraction, so no thread holds or is waiting for the sleeplock, which is exactly the precondition of the non-blocking specification of `acquiresleep`. Acquiring the sleeplock then hands `iput` a fresh fraction of the same size, which restores the total in the lock invariant of `itable.lock` to what it was, so that `iput` can release `itable.lock` afterwards.

6.8 Process exit

When a process exits in xv6, the kernel must reclaim ownership of the process's kernel stack. However, the code path of the process exiting involves a trap vector at the top level, `uservec`, invoking the C-level trap handler `usertrap`, which calls `syscall`, which in turn dispatches `sys_exit`, and ultimately this sets the process state to `ZOMBIE` and switches to the scheduler. The challenge is that, at each level of this call stack, the callers (such as `usertrap` and `syscall`) still have CSL-level ownership of resources representing their parts of the stack, corresponding to their stack variables, saved registers, etc. At the same time, in order to reuse the kernel stack of this exited process, the kernel needs to regain ownership of the entire 4 KB page of memory.

The xv6 proof addresses this challenge by extending the specification of the calling convention with explicit support for function calls that might never return. In particular, if a function chooses not to return to its caller, its caller should give up ownership of its stack memory locations that might be implicitly captured inside the caller's continuation wp CpuLoop. We encode this using separation logic's $\wedge$ operator (AND). Unlike the $*$ separating conjunction, where $A * B$ means that A and B exist separately, and ownership of $A * B$ allows using both of them, $A \wedge B$ means that A and B are both true but they might overlap, so they are not facts about disjoint parts of the state. The rule for using $A \wedge B$, then, is that a proof that owns $A \wedge B$ must choose which one to get: either get ownership of A, but that makes B disappear, or get ownership of B, but that makes A disappear.

Figure 12 shows this pattern in a simplified version of the spec for the `syscall` function. With this specification, the proof of `syscall` can choose to either use the left side of the $\wedge$, and return to the caller, or use the right side of the $\wedge$, whereby the stack_reclaim m[sp] resource allows it to reclaim ownership of all stack memory from the current stack pointer up towards the top of the stack (sie_cap_gpr holds ownership of the other half of the stack, namely, all of the free stack locations below the current stack pointer). The proof of the caller of `syscall`, namely `usertrap`, has to supply both of these resources to `syscall`'s proof. A function's proof may need to selectively invoke its own caller's WP or stack_reclaim resources to satisfy these two obligations when calling a function that might not return.

$$
\begin{array}{l}
\mathsf{kernel_text} \mathrel{-\!\!*} \\
\mathtt{pc} \mapsto_{\mathsf{Reg}} \mathtt{syscall} \mathrel{-\!\!*} \\
\mathsf{sie_cap_gpr}\ m\ n\ \ldots \mathrel{-\!\!*} \\
\ldots \mathrel{-\!\!*} \\
(\ (\forall\, m'.\ \ \mathtt{pc} \mapsto_{\mathsf{Reg}} m[\mathtt{ra}] \mathrel{-\!\!*} \\
\qquad\quad \mathsf{sie_cap_gpr}\ m'\ n\ \ldots \mathrel{-\!\!*} \\
\qquad\quad \mathsf{callee_saved}\ m = \mathsf{callee_saved}\ m' \mathrel{-\!\!*} \\
\qquad\quad \ldots \mathrel{-\!\!*} \\
\qquad\quad \mathsf{wp\ CpuLoop}) \wedge \\
\quad \mathsf{stack_reclaim}\ m[\mathtt{sp}]\ ) \mathrel{-\!\!*} \\
\mathsf{wp\ CpuLoop}
\end{array}
$$

Figure 12: Simplified spec for the `syscall` function, demonstrating how the specification captures the possibility of `syscall` not returning, and in that case, being able to reclaim ownership of the caller's stack frame.

For functions that never return, such as `sys_exit`, the specification does not require any wp CpuLoop continuation; the caller must always supply just the stack_reclaim resource. At the bottom level, the specification of `swtch` requires the caller to supply either a wp CpuLoop for returning, which is placed into a valid_context for eventually resuming the kernel thread, or a stack_reclaim resource that allows the scheduler to reclaim the entire kernel stack. In particular, the xv6 scheduler specification requires the caller to supply the wp CpuLoop continuation if the process is `RUNNABLE` or `SLEEPING`, and supply the stack_reclaim resource if the process is `ZOMBIE`.

6.9 Compiler optimizations

One advantage of verifying the kernel at the RISC-V level in MachCSL is that the proof subsumes any compiler optimizations. One surprising optimization that the xv6 proof ran into is GCC's computation of a pointer difference within a struct array. In particular, the `procinit` function initializes the kernel stack for all `struct procs`, at kernel boot time, as shown in Figure 13. The `p - proc` expression, in C, computes the index of pointer `p` in the `proc` array, which requires dividing the difference between the two pointer addresses by `sizeof(struct proc)`, which happens to be 360.

```
struct proc {
  ...
  uint64 kstack;  // Virtual address of kernel stack
  ...
};

struct proc proc[NPROC];

#define KSTACK(p) (TRAMPOLINE - ((p) + 1) * 2 * PGSIZE)

void
procinit(void)
{
  struct proc *p;

  for (p = proc; p < &proc[NPROC]; p++) {
    ...
    p->kstack = KSTACK((int)(p - proc));
  }
}
```

Figure 13: Example of code from the `procinit` function in xv6 that triggers a surprising optimization when compiled with GCC.

When GCC compiles this code, it avoids emitting what would be the direct translation using a divide-by-360 instruction. Instead, GCC's optimization first observes that $360 = 8 \times 45$, so to divide by 360, it suffices to divide by 8 (by doing a right-shift by 3), and then divide by 45. Then, to divide by 45, GCC computes the modular inverse of 45, $M = 45^{-1} \bmod 2^{64} = \texttt{0x4fa4fa4fa4fa4fa5}$. Finally, since GCC knows that the difference between the pointer addresses `p` and `proc`, which we will call δ, is divisible by 360, and thus that $\frac{\delta}{8}$ is divisible by 45 and less than 2^{63}, dividing it by 45 is equivalent to multiplying by M. Thus, GCC computes $\texttt{p} - \texttt{proc} = \frac{\delta}{360}$ as $(\delta \gg 3) \times M$. The proof of `procinit` in xv6 proves the correctness of this modular-inverse optimization.

6.10 UART output

The specification for `uartwrite`, used in xv6 to print output to the console, is stated in terms of a ghost variable tracking an append-only list of bytes printed to the console, uart_sent γ *bs*. The specification for `printk`, which we will discuss next, says that, whatever the console output was before, the result will be that the console now contains the output of `printk` as well. Since the console output is append-only, `printk` can return this fact in its postcondition as a fact about a *prefix* of the console output; more bytes might have been written by other processes, or by `printk` calls on other cores, but `printk`'s output is guaranteed to be present.

6.11 Specifying `printk`

xv6 implements an in-kernel variant of `printf`, which it calls `printk`. Specifying `printk` is challenging because it takes a variable number of arguments, and because the facts that the caller's proof has to establish about each of those arguments depend on the contents of the format string passed to `printk` in the first argument.

Figure 14 shows a simplified version of the specification for `printk` in xv6. The pure function kinds walks the format string f and returns, in order, the kind of every argument that `printk` will consume: a Num for the integer conversions (`%d`, `%u`, `%x`, `%p`, `%c`, and their `l` and `ll` variants, which the figure elides), and a Str for `%s`. Escapes such as `%%` and unrecognized conversions consume no argument, exactly as in the C code. Since kinds is a structural fixpoint, Rocq proofs can compute its result—for example, for the format string used by `syscall` to report an unknown system call, kinds `"%d %s: unknown sys call %d\n"` = [Num; Str; Num] by reflexivity.

The resource that the caller must supply for each kind, kind_res, is where the argument types differ. An integer, character, or pointer printed with `%p` costs nothing: in RISC-V, every variadic argument is passed in a 64-bit register, `printk` only reads the value of a Num, and its integer-printing routine can handle any value. The C-level distinctions between `int`, `long`, and `char` are invisible to the logic because they are invisible at the instruction level. For strings (`%s`), the caller's proof must either prove that the pointer is null (in which case `printk` prints `(null)`), or supply a string points-to resource $v \mapsto_{\mathsf{vMemString}} s$ for the argument's value v.

args combines all of the required resources, given the format string f, and also encodes the calling convention: argument j lives in register a(j+1). The bound $|\mathsf{kinds}\ f| \le 7$ encodes the assumption that `printk` gets no more than 7 arguments; relaxing this assumption would require extending args to know about how the calling convention passes further arguments on the stack. The postcondition returns the resources back to the caller, and also says that some byte list cs has been appended to the bytes that this caller has provably sent to the UART. The spec does not precisely pin down the resulting byte sequence sent to the UART, since the exact format of kernel `printk` debug messages was not important for our verification goal.

$$
\begin{array}{l}
\mathsf{kind} \triangleq \mathsf{Num} \mid \mathsf{Str} \\
\mathsf{kinds} : \mathit{string} \to \mathit{list}\ \mathsf{kind} \\
\quad \mathsf{kinds}\ (\texttt{"\%d"} +\!\!+ r) = \mathsf{Num} :: \mathsf{kinds}\ r \\
\quad \mathsf{kinds}\ (\texttt{"\%s"} +\!\!+ r) = \mathsf{Str} :: \mathsf{kinds}\ r \\
\quad \mathsf{kinds}\ (\texttt{"\%\%"} +\!\!+ r) = \mathsf{kinds}\ r \\
\quad \mathsf{kinds}\ (c :: r) = \mathsf{kinds}\ r \quad \text{(any other } c\text{)} \\
\quad \mathsf{kinds}\ \texttt{""} = [] \\
\\
\mathsf{kind_res}\ v\ \mathsf{Num} \triangleq \mathsf{True} \\
\mathsf{kind_res}\ v\ \mathsf{Str} \triangleq v = 0 \lor \exists\ s.\ v \mapsto_{\mathsf{vMemString}} s \\
\\
\mathsf{args}\ m\ f \triangleq \\
\quad \mathop{*}_{j \mapsto k \in \mathsf{kinds}\ f} \mathsf{kind_res}\ m[\mathsf{a}(j{+}1)]\ k
\end{array}
$$

$$
\begin{array}{l}
|\mathsf{kinds}\ f| \le 7 \to \\
\mathsf{kernel_text} \mathrel{-\!\!*} \\
\mathsf{pc} \mapsto_{\mathsf{Reg}} \mathtt{printk} \mathrel{-\!\!*} \\
\mathsf{sie_cap_gpr}\ m\ \ldots \mathrel{-\!\!*} \\
m[\mathtt{a0}] \mapsto_{\mathsf{vMemString}} f \mathrel{-\!\!*} \\
\mathsf{args}\ m\ f \mathrel{-\!\!*} \\
\mathsf{uart_sent}\ \gamma\ bs \mathrel{-\!\!*} \ldots \mathrel{-\!\!*} \\
(\ \forall\ m'\ cs.\ \mathsf{pc} \mapsto_{\mathsf{Reg}} m[\mathtt{ra}] \mathrel{-\!\!*} \\
\qquad \mathsf{sie_cap_gpr}\ m' \ \ldots \mathrel{-\!\!*} \\
\qquad \mathsf{callee_saved}\ m = \mathsf{callee_saved}\ m' \land \\
\qquad\quad m'[\mathtt{a0}] = 0 \mathrel{-\!\!*} \\
\qquad m[\mathtt{a0}] \mapsto_{\mathsf{vMemString}} f \mathrel{-\!\!*} \\
\qquad \mathsf{args}\ m\ f \mathrel{-\!\!*} \\
\qquad \mathsf{uart_sent}\ \gamma\ (bs +\!\!+ cs) \mathrel{-\!\!*} \\
\qquad \mathsf{wp}\ \mathsf{CpuLoop}\ ) \mathrel{-\!\!*} \\
\mathsf{wp}\ \mathsf{CpuLoop}
\end{array}
$$

Figure 14: Simplified spec for `printk`.

6.12 Bounded physical memory

The implementation of `exec` in xv6, the `kexec` function, initializes the page table of the fresh process based on the supplied ELF image of the user binary that is being `exec`'ed. This is tricky code because it populates the page table based on an untrusted ELF image. In particular, consider the simplified snippet of this code shown in Figure 15. `kexec` passes arbitrary values from the ELF program header `ph` as the `newsz` argument of `uvmalloc`, and `uvmalloc`'s call to `mappages` will panic if it turns out this address is already mapped. Every user page table has a trampoline mapping at the very top of the virtual address space, used to switch between user and kernel page tables, and thus the proof had to consider whether `kexec` could possibly call `panic` on some adversarial ELF binary.

The reason this panic turns out to be unreachable is that, in order for `uvmalloc`'s for loop to reach the trampoline virtual address (which would have triggered the panic), it has to first allocate physical memory pages all the way up to that high virtual address. This is not possible; the system simply does not have that much physical memory. Thus, the kernel proof is able to conclude that the panic is unreachable, by contradiction: if the execution reached that panic, it would mean that the current proof holds separation-logic resources for more physical memory than is possible on the machine.

6.13 Safety of arbitrary user-mode execution

An OS kernel must ensure that, when it starts executing processes in user mode, those processes cannot corrupt the execution of the kernel. The xv6 kernel proof handles this by proving a WP theorem about execution in user mode. Figure 16 shows a simplified version of this theorem. Unlike every other wp CpuLoop specification, this spec does not talk about execution at a specific PC address. Instead, it covers execution at any PC, as long as the processor remains at the user privilege level, for an arbitrary number of cycles.

```
int
kexec(char *path, char **argv)
{
  ...
  for (i = 0; i < elf.phnum; i++) {
    readi(ip, 0, (uint64)&ph, off, sizeof(ph));
    ...
    if (ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz);
    ...
  }
  ...
}

uint64
uvmalloc(pagetable_t pt, uint64 oldsz, uint64 newsz)
{
  for (a = oldsz; a < newsz; a += PGSIZE) {
    mem = kalloc();
    if (mem == 0) {
      uvmdealloc(pt, a, oldsz);
      return 0;
    }
    ...
    if (mappages(pt, a, PGSIZE, (uint64)mem) < 0) {
      ...
      return 0;
    }
  }
  return newsz;
}

int
mappages(pagetable_t pt, uint64 va,
         uint64 sz, uint64 pa)
{
  ...
  pte = walk(pt, va, 1);
  if (pte == 0)
    return -1;
  if (*pte & PTE_V)
    panic("mappages: remap");
  ...
}
```

Figure 15: Simplified code from `kexec`, `uvmalloc`, and `mappages` in xv6.

The user-mode execution invariant, user_inv, describes what it means for the system to be correctly executing user-mode code: the current privilege is `User`; the HART is either active or waiting (sleeping); the program counter and the general-purpose registers have arbitrary contents; and `mstatus` is arbitrary up to a few pinned bits that user execution cannot change (for instance, `MPRV` and `MXR` are clear, so the user program's loads and stores are translated as user accesses). The supervisor trap CSRs (`scause`, `stval`, `sepc`) hold arbitrary values; they are updated when the processor takes a trap.

The invariant also owns the user page table, user_pt *pt*: the `satp` register, the page-table pages that make up the tree rooted at *pt.root*, and $pa \mapsto_{\mathsf{Mem}} b$ for every byte of every physical page mapped with the user bit set, with arbitrary contents. The domain of this memory map is pinned to exactly the mapped virtual addresses, so user_pt says "this is *all* of the user's memory", and no other part of the kernel proof can own any of it. The pure side conditions, abbreviated pt_wf, say that the translation is injective (no two user virtual addresses map to the same physical byte, so a store through one address cannot invalidate the points-to for another) and that the page-table

```
user_cfg C ≜
  stvec ↦Reg C.stvec ∗ medeleg ↦Reg C.medeleg ∗ ... ∗
  direct_mode C.stvec ∗
  user_exceptions_delegated C.medeleg ∗ ...

user_pt pt ≜
  satp ↦Reg pt.root ∗ pt_pages pt ∗ pt_wf pt ∗
  ∃ M. dom M = mapped pt ∗
       ∗_{va↦b∈M} (translate pt va) ↦Mem b

user_inv C pt ≜ ∃ hs ms pc m c t e.
  cur_privilege ↦Reg User ∗
  hart_state ↦Reg hs ∗ hs ∈ {Active, Waiting} ∗
  mstatus ↦Reg ms ∗ user_mstatus_ok ms ∗
  pc ↦Reg pc ∗ gpr_file m ∗
  scause ↦Reg c ∗ stval ↦Reg t ∗ sepc ↦Reg e ∗
  user_pt pt ∗ user_cfg C

user_trap_frame C pt ≜ ∃ ms m c t e.
  cur_privilege ↦Reg Supervisor ∗
  hart_state ↦Reg Active ∗
  mstatus ↦Reg ms ∗ trap_mstatus_ok ms ∗
  pc ↦Reg C.stvec ∗ gpr_file m ∗
  scause ↦Reg c ∗ stval ↦Reg t ∗ sepc ↦Reg e ∗
  user_pt pt ∗ user_cfg C

user_inv C pt −∗
(user_trap_frame C pt −∗ wp CpuLoop) −∗
wp CpuLoop
```

Figure 16: Simplified spec for arbitrary user-mode execution. The user-mode execution invariant user_inv owns the entire machine state a user instruction can touch, with all mutable parts existentially quantified; the only way out is a trap that delivers user_trap_frame to the kernel's handler at stvec.

entries are well-formed. CSL's exclusive ownership rules mean that the physical pages that are in the user's page table are disjoint from any other state owned by the kernel.

Finally, the invariant holds the configuration parameters in user_cfg C: ownership of stvec and of the delegation registers, together with the facts that stvec is in direct mode (so every trap lands exactly at its base address), that every exception that user code can raise is delegated to supervisor mode, and that no interrupt is enabled that would be handled in machine mode. These last facts are what guarantee that a trap out of user mode lands at the kernel's handler rather than somewhere the proof knows nothing about.

The specification then says that, given user_inv and a wp CpuLoop for the trap handler, the machine may run forever. The trap-handler wp CpuLoop receives the user_trap_frame resource, which is what a trap from user mode hands the kernel: privilege Supervisor, the PC at C.stvec, the trap CSRs freshly written, and the same page table and configuration as before the trap. Note that scause, stval, and sepc are existential in the trap frame, since the handler must handle every kind of trap; the per-cause lemmas that produce this frame know their exact values.

The proof itself is an induction over any execution that starts at the user privilege level. At each cycle, it considers the possibility that the user code might trap, jumping to the trap handler (which is proven by using the WP of the trap handler address), or it might execute arbitrary code. The theorem considers every possible outcome of executing the RISC-V fetch phase, followed by an arbitrary decode of the fetched instruction bytes, followed by an arbitrary execution of the decoded instruction. For every single one of these cases, the proof concludes that either user_inv is re-established, and the proof continues by induction, or the machine has trapped and user_trap_frame holds, and the proof concludes with the trap handler's WP.

One surprise we discovered in the process of doing this user-mode inductive proof is that user-mode code in RISC-V can put the core to sleep. At first, the user-mode execution invariant included the fact that the RISC-V core

running the user code is active (i.e., not sleeping). As it turns out, there is a RISC-V instruction, `WRS.NTO`, that can put the CPU to sleep while executing in user mode. This is safe because interrupts are still enabled, so the kernel will safely regain execution at the next timer interrupt. However, this required us to extend the user-mode invariant to allow being at user privilege level while the core is actually sleeping.

6.14 Visibility of writes under TSO

A key challenge that TSO introduces is that different cores can see different results when reading the same memory location, because of store buffering. There are several places where this shows up in an interesting way in the xv6 proof, as we now describe.

One example is shared state protected by a spinlock. When one core releases a lock, it passes ownership of the shared state to the spinlock's lock invariant. When another core acquires the lock, it should get ownership of the same lock invariant, but in the proof, the lock invariant was originally true on the first core, where it might have referred to writes in that core's store buffer. The proof of the second core, however, must now establish that those same writes are visible on the second core as well.

A second example is the xv6 spinlock implementation: it attempts to guard against deadlock by checking whether the current CPU already holds the spinlock before trying to acquire it. This `holding` check does a non-atomic read of the spinlock's `lk->cpu` field, which stores a pointer to the `struct cpu` of the CPU holding this lock, if any. However, under TSO, the read is not guaranteed to see the latest value of this field, which makes it difficult for the proof to establish that this `holding` check will never panic—after all, it might be that, at some point in the past, this CPU did hold the lock.

A third example is context switching of processes between cores: when an interrupt comes in, a running kernel thread gets suspended, and another CPU's scheduler may choose to resume running it on another core. However, under TSO weak memory, different cores might have different views of shared memory. This makes it non-trivial to prove that it is safe to resume the suspended kernel thread on a different core.

To address this, the xv6 proof introduces a notion of a *view*, which logically captures the view of memory from the perspective of some core. Any CSL resources in the xv6 proof that are intended to be shared across cores are stated as being view-dependent (that is, the resource is a function of the view). The specification of a spinlock associates a logical view with the spinlock itself—this captures the notion that the lock invariant is true with respect to some kind of logical spinlock view, rather than the view of any particular core. The lock invariant pins down exactly what this view means: it is the view corresponding to the timestamp of the latest `release` write to the lock word. This allows the `acquire` proof to establish that, if `acquire` saw the lock in an unlocked state, its local CPU timestamp must be high enough to see all of the writes in the lock's view. This in turn allows `acquire`'s proof to hand back the lock invariant to the caller's proof relative to the caller's own view on the local CPU.

Using views, the xv6 proof also introduces a resource to represent non-sequentially-consistent shared memory locations, $a \mapsto_\xi \{S\}$. This resource says that, in view ξ, reading address a can result in any value from set S. This is effectively a statement about a suffix of the history of address a starting from the timestamp associated with view ξ. This allows us to capture the invariant for the lock's `lk->cpu` field: namely, `lk->cpu` $\mapsto_\xi \{S\}$ where S cannot contain this CPU. The view ξ is important for dynamically allocated spinlocks. In particular, xv6 dynamically allocates pages of memory for pipes, which contain a spinlock, and the proof must establish that, when acquiring a pipe spinlock, the `holding` check does not inadvertently see an old value that happened to be at `lk->cpu` from before this memory location was reused for a spinlock.

Finally, to reason about context switching of kernel threads, the xv6 proof introduces the notion of suspending one view under another view. xv6 has a kernel thread for every process, and also a kernel thread for every CPU's scheduler. The per-process kernel threads migrate between cores, but the scheduler kernel threads are pinned to their respective cores. When a process kernel thread context-switches into the scheduler, the scheduler's proof suspends the process thread's view under its own view. This means that the set of writes visible to the process thread is a subset of the writes visible to the scheduler thread. The scheduler then releases the lock associated with the process, `p->lock`, which moves the suspended view from being relative to the scheduler thread's view to being suspended under the spinlock's view. When another core acquires `p->lock` to run that process, the lock invariant gives it ownership of the suspended process view, now under the new scheduler thread's view. Finally, when the new scheduler jumps to the resumed kernel thread, it un-suspends the process view.

View-relative predicates also show up in other places where resources move across cores. For example, in xv6, the first CPU is responsible for initializing many kernel subsystems, including the memory allocator, the page table, etc. The invariants that the first CPU establishes about these subsystems are themselves view-relative, which means

they might not be visible to other cores yet. xv6 uses a barrier variable, `started`, that other cores poll to see if the first core has finished initializing yet. The invariant associated with `started` is $\texttt{started} \mapsto_0 \{0, 1\}$, which means that it can only have the values 0 and 1, starting from the system boot time, along with a predicate that says that, if the value is 1, the reader can learn the view-relative resources initialized by the first CPU.

6.15 Interrupt exclusivity

The UART driver in xv6 relies on the PLIC (platform-level interrupt controller) to ensure that only one core is handling UART receive interrupts at any given time, to avoid race conditions in accessing the UART's receive FIFO. The PLIC on RISC-V provides a way for a core to "claim" an interrupt, which both returns the IRQ of the pending interrupt and prevents other cores from claiming it until released. The simplified specifications for these PLIC functions in xv6, `plic_claim` and `plic_complete`, are shown in Figure 17. When `plic_claim` returns an interrupt, it also returns exclusive resources intended for that interrupt's handler. In the case of the UART, it is the uart_rx_tok token that allows the proof of the UART receive interrupt handler to prove that no other cores are accessing the receive FIFO at the same time.

$$
\begin{array}{l}
\text{kernel_text} \mathrel{-\!\!*} \\
\texttt{pc} \mapsto_{\text{Reg}} \texttt{plic_claim} \mathrel{-\!\!*} \\
\text{sie_cap_gpr } m\ n \text{ false } \ldots \mathrel{-\!\!*} \\
\text{plic_inv } \gamma \mathrel{-\!\!*} \\
(\ \forall\ m'.\ \texttt{pc} \mapsto_{\text{Reg}} m[\texttt{ra}] \mathrel{-\!\!*} \\
\qquad \text{sie_cap_gpr } m'\ n \text{ false } \ldots \mathrel{-\!\!*} \\
\qquad \text{callee_saved } m = \text{callee_saved } m' \wedge \\
\qquad\quad m'[\texttt{a0}] \in \{0, \text{UART_IRQ}, \text{VIRTIO_IRQ}\} \mathrel{-\!\!*} \\
\qquad (m'[\texttt{a0}] = \text{UART_IRQ} \mathrel{-\!\!*} \exists\ k.\ \text{uart_rx_tok } \gamma\ k) \mathrel{-\!\!*} \\
\qquad \text{wp CpuLoop }) \mathrel{-\!\!*} \\
\text{wp CpuLoop}
\end{array}
\qquad
\begin{array}{l}
\text{kernel_text} \mathrel{-\!\!*} \\
\texttt{pc} \mapsto_{\text{Reg}} \texttt{plic_complete} \mathrel{-\!\!*} \\
\text{sie_cap_gpr } m\ n \text{ false } \ldots \mathrel{-\!\!*} \\
\text{plic_inv } \gamma \mathrel{-\!\!*} \\
m[\texttt{a0}] \in \{0, \text{UART_IRQ}, \text{VIRTIO_IRQ}\} \mathrel{-\!\!*} \\
(m[\texttt{a0}] = \text{UART_IRQ} \mathrel{-\!\!*} \exists\ k.\ \text{uart_rx_tok } \gamma\ k) \mathrel{-\!\!*} \\
(\ \forall\ m'.\ \texttt{pc} \mapsto_{\text{Reg}} m[\texttt{ra}] \mathrel{-\!\!*} \\
\qquad \text{sie_cap_gpr } m'\ n \text{ false } \ldots \mathrel{-\!\!*} \\
\qquad \text{callee_saved } m = \text{callee_saved } m' \mathrel{-\!\!*} \\
\qquad \text{wp CpuLoop }) \mathrel{-\!\!*} \\
\text{wp CpuLoop}
\end{array}
$$

Figure 17: Simplified PLIC specifications for `plic_claim` and `plic_complete`.

The specification allows multiple cores to concurrently invoke `plic_claim`. By promising to return the exclusive resources of the UART for the UART IRQ, the specification promises that it will not return the UART IRQ to a second core until the first core releases it using `plic_complete`. The proof of `plic_claim` works by opening an invariant at the instant it performs the MMIO access to the PLIC's claim register, similar to the proof of `acquire`.

7 Proving user-level code

The previous section describes the specification and proof of the xv6 kernel itself, including the safety of executing arbitrary user-level code. This suffices to conclude that user-level programs cannot corrupt the kernel, and all kernel state will remain intact (e.g., the file system remains consistent and no invariants are violated).

This section describes how we extend specifications and proofs to allow proving the correctness of user-level programs running on top of the xv6 kernel. For example, we would like to prove our application theorem from §2, which implies that if the user types `echo hello world` on the UART console, then the only response the user should see is `hello world` printed back to them (or one of the allowed error messages). Proving such a theorem requires reasoning about the correctness of multiple user-level processes, including `init` (which is responsible for setting up the stdin, stdout, and stderr file descriptors and launching the shell), `sh` (which is responsible for taking user input from the console, parsing it, and running commands), and `echo` (which prints its string arguments from `argv`), as well as precisely reasoning about the interaction between those processes and the kernel.

This theorem is succinct—it is simply a statement about the sequence of bytes sent on the UART console—yet it implies that a wide range of problems cannot occur. For example, it implies that the file system must contain the correct binaries for `/init`, `/sh`, `/echo`, etc., across any possible sequence of reboots. It also implies that the kernel correctly implements all of the system calls, including `fork`, `exec`, `read`, `write`, `wait`, and `open`. It also implies that the user-level programs themselves are correct—e.g., `init` does not spawn multiple shells that might race for reading input from the console, and `sh` correctly runs `echo`. It also means there are no memory safety violations—the stack of each user-level process never overflows, taking into account the initial `argv` placed on the user-level stack by `exec`. Finally, this also implies that the kernel correctly executes user-level processes, such as ensuring that the instruction cache is flushed when a process is migrated from one core to another.

An important aspect of our approach to proving user-level programs is that the kernel's specification and proof remain generic. In order to prove the correctness of the `echo hello world` application in our example above, it suffices to prove theorems about the user-level programs involved in orchestrating this execution (`init`, `sh`, and `echo`), but the kernel's specifications and proofs can be used as-is.

7.1 User-mode execution environment

To enable proofs that reason about the execution of user-level code, the xv6 kernel's specification defines the complete state of a user-level process, in terms of what the user-level code can observe. This is captured by the uvis record as shown in Figure 18. All user-level execution, including instructions, memory accesses, and system calls, is defined in terms of how it accesses and updates this uvis state.

```
uvis ≜ {
  tf : list word;          trapframe: registers and PC to resume at
  M : vaddr ⇀ byte;        memory image
  π : page ⇀ perm;         per-page R/W/X permissions
  sz : Z;                  program break
  fds : list fdstate;      one state per file descriptor
  cwd : inum;              working directory
  pid : Z;                 process ID
  gen : gname;             this process's generation
  ch : set gname           generations of live children
}
```

Figure 18: A simplified version of the user-visible state of a process, uvis.

uvis contains the trapframe words, representing the register contents when the process enters the kernel; the process's memory image M; the per-page permission view π; the program break sz; the state of each file descriptor (closed, or open at a particular file); the current working directory; etc. The user-mode process can observe and modify this state both through explicit system calls (e.g., `getpid` returns the process's own PID) and through executing RISC-V instructions (e.g., executing code at a particular program counter `pc` depends on whether that page is mapped executable or not according to the π permission table). The user-visible state does not explicitly list the page table, or the current CPU on which the process is executing, because those are transparently managed by the kernel, and the execution of the user process can be described just in terms of the uvis state.

Separation logic for user-mode memory. Reasoning directly about execution on top of uvis can be cumbersome because that would require global specs: for example, every instruction that touches memory would have to say what happens to every location in memory. To get the modularity advantages of CSL for user-level code, we developed a user-level proof library that builds up separation-logic abstractions on top of the raw uvis state.

In particular, the user-level proof library creates two separation-logic heaps that represent the raw memory image M: one corresponding to the executable text of the process, and one corresponding to the rest of the process memory. Text bytes, $a \mapsto_{\mathsf{uText}\,\gamma} b$, are persistent, because xv6 processes never modify their code; holding one is the right to fetch an instruction from a. The fact that text pages are read-only ensures that the HART's icache is consistent with memory, and allows the proof to conclude that fetching from that address will obtain the correct instructions (as well as conclude that no page fault will occur due to a missing page-execute bit). Data bytes, $a \mapsto_{\mathsf{uData}\,\gamma} b$, are exclusive, and holding one is the right to write a. Read-only views of data bytes, $a \mapsto^{\mathsf{ro}}_{\mathsf{uData}\,\gamma} b$, are used for read-only state, such as static strings.

All user-mode proofs keep around a urun resource, as shown in Figure 19, that maintains a correspondence between the underlying uvis state and the separation-logic view of the state used in the user-level proofs. The uheap predicate ties the two separation-logic heaps, $\gamma.t$ and $\gamma.d$, to the memory image M. The part of the last page of memory above the program break is owned by urun and not exposed in the data heap.

The user-level proof library defines specifications for each RISC-V instruction that user-level code can execute in terms of the separation-logic state exposed by urun. For example, the specification for `ADDI` needs only the instruction's text bytes, uinstr γ pc i (a persistent fact derived from $\mapsto_{\mathsf{uText}\,\gamma}$), and urun, as shown in Figure 20. Similarly, the specification for `SD` additionally consumes and returns the heap points-to resource for the word being overwritten.

One detail that the `SD` specification abstracts over is the fact that the xv6 kernel implements lazy allocation of zeroed pages. The uvis state represents such pages as containing all zeroes, even though there might be no physical

$$\begin{aligned}
&\mathsf{uheap}\ \gamma\ M\ \pi\ sz \triangleq \exists\, M_t\ M_d\ M_{slack}.\\
&\quad M_t \subseteq M * M_d \subseteq M * M_t \,\#\, M_d *\\
&\quad \mathsf{dom}\ M_t \subseteq \mathsf{exec_only}\ \pi * \mathsf{dom}\ M_d \subseteq \mathsf{writable}\ \pi *\\
&\quad \mathsf{auth}_{\gamma.t}\ M_t * \mathsf{auth}_{\gamma.d}\ M_d * \mathsf{usz}_{\gamma.s}\ sz *\\
&\quad M_{slack} = \{a \mapsto b \in M_d \mid a \geq sz\} * \mathop{*}_{a \mapsto b \in M_{slack}} a \mapsto_{\mathsf{uData}\,\gamma} b
\end{aligned}$$

$$\begin{aligned}
&\mathsf{urun}\ \gamma\ m\ pc\ n \triangleq \exists\, M\ \pi\ sz\ fds\ cwd\ pid\ gen\ ch\ pt\ C.\\
&\quad \mathsf{uheap}\ \gamma\ M\ \pi\ sz * \mathsf{ustack}\ \gamma\ m[\mathsf{sp}]\ n *\\
&\quad \mathsf{auth}_{\gamma.fd}\ fds * \mathsf{auth}_{\gamma.cwd}\ cwd * \mathsf{auth}_{\gamma.ch}\ ch *\\
&\quad \mathsf{ukernel}\ pt\ C\ (\mathsf{uvis_of}\ m\ pc\ M\ \pi\ sz\ fds\ cwd\ pid\ gen\ ch)
\end{aligned}$$

Figure 19: Simplified definitions of the per-process user heap and running-process predicate urun. γ is a record of the process's ghost names.

$$\begin{aligned}
&rd \notin \{\mathtt{x0}, \mathsf{sp}\} \rightarrow\\
&\mathsf{uinstr}\ \gamma\ pc\ (\mathtt{ADDI}\ rd, rs1, imm) -\!\!*\\
&\mathsf{urun}\ \gamma\ m\ pc\ n -\!\!*\\
&(\ \mathsf{urun}\ \gamma\ m[rd := m[rs1] + imm]\\
&\qquad (pc + 4)\ n -\!\!*\\
&\ \ \mathsf{wp\ CpuLoop}\ ) -\!\!*\\
&\mathsf{wp\ CpuLoop}
\end{aligned}$$

$$\begin{aligned}
&a = m[rs1] + imm \rightarrow a \bmod 8 = 0 \rightarrow\\
&\mathsf{uinstr}\ \gamma\ pc\ (\mathtt{SD}\ rs2, imm(rs1)) -\!\!*\\
&a \mapsto_{\mathsf{uWord}\,\gamma} v -\!\!*\\
&\mathsf{urun}\ \gamma\ m\ pc\ n -\!\!*\\
&(\ a \mapsto_{\mathsf{uWord}\,\gamma} m[rs2] -\!\!*\\
&\quad \mathsf{urun}\ \gamma\ m\ (pc + 4)\ n -\!\!*\\
&\quad \mathsf{wp\ CpuLoop}\ ) -\!\!*\\
&\mathsf{wp\ CpuLoop}
\end{aligned}$$

Figure 20: User-mode instruction rules for `ADDI` (left) and `SD` (right), stated on urun. m is the register file of urun.

page allocated for that range of memory. When an `SD` instruction stores into a page of memory (or when an `LD` instruction loads from memory), the process might take a page fault if the page has not been allocated yet. The kernel allocates a fresh page, zero-fills it, installs it into the page table, and resumes the execution of the process, with the same exact visible state according to uvis. After the page has been installed, the `SD` or `LD` instruction can resume execution.

7.2 System calls

xv6 processes use the `ECALL` instruction to invoke a system call. The kernel provides a specification for a trap caused by `ECALL`, in terms of how it affects uvis, and the user-level proof library builds on that specification to provide an equivalent separation-logic-based spec for `ECALL`. Instead of specifying the complete behavior of `ECALL` for all possible system calls that might be invoked, the user-level proof library provides separate specifications for each system call number.

File descriptor state. The state of each file descriptor is part of uvis as a list of fdstates, each of which is either Closed or Open $r\ w\ t$, where r and w are the readable and writable flags and the type t says whether the descriptor is an inode (with its inode number, and the ghost name γ_o of its offset), a pipe end, or a device (such as the UART console).

The user-level proof library introduces separation-logic resources to track the state of each file descriptor. There are two kinds of such resources, to represent the Unix convention that file descriptors 0, 1, and 2 are stdin, stdout, and stderr respectively. These three lowest-numbered file descriptors are represented by ustd $\gamma\ l$, and higher-numbered file descriptors are represented by ufd $\gamma\ fd\ st$. A wrapper ufd_own defines a valid file descriptor that is either low- or high-numbered.

Specifying `dup`. Figure 21 shows the specification for the `dup` system call, which clones an existing file descriptor, in terms of these separation-logic resources. The rule for `dup` takes the resource of the file descriptor being cloned, and returns either a new descriptor allocated at the lowest closed slot with a copy of the source's *state* (so the program knows the new descriptor has the same type and mode as the source), or -1 with everything unchanged. The special treatment of low-numbered file descriptors allows proving `init`'s code that sets up stdin, stdout, and stderr, by calling `open(...)` to open file descriptor 0, and then calling `dup(0)` twice to clone it as file descriptors 1 and 2.

7.3 File system state

To enable user-level proofs to reason about file system state, the xv6 kernel specification exposes an abstract view of the file system state, as shown in Figure 22. This abstract view is used to specify the state of the file system in memory, as well as the on-disk durable state of the file system that will be present if the system crashes and reboots.

$$
\begin{array}{l}
\mathsf{fdstate} \triangleq \mathsf{Closed} \mid \mathsf{Open}\ (r\ w : \mathit{bool})\ (t : \mathsf{fdtype}) \\
\mathsf{fdtype} \triangleq \mathsf{Inode}\ \mathit{inum}\ \gamma_o \mid \mathsf{Pipe} \mid \mathsf{Device}\ \mathit{major} \\
\mathsf{ufd}\ \gamma\ \mathit{fd}\ \mathit{st} \triangleq \mathit{fd} \hookrightarrow_{\gamma.\mathit{fd}} \mathit{st} * \mathit{st} \neq \mathsf{Closed} * 3 \le \mathit{fd} \\
\mathsf{ustd}\ \gamma\ l \triangleq |l| = 3 * \mathop{\Large *}_{k<3} k \hookrightarrow_{\gamma.\mathit{fd}} l[k] \\
\mathsf{ufd_own}\ \gamma\ l\ \mathit{fd}\ \mathit{st} \triangleq (\mathit{fd} < 3 \wedge l[\mathit{fd}] = \mathit{st}) \vee \mathsf{ufd}\ \gamma\ \mathit{fd}\ \mathit{st} \\
\mathsf{ualloc}\ \gamma\ l\ \mathit{fd}\ \mathit{st} \triangleq \mathsf{ustd}\ \gamma\ (\mathsf{fill}\ l\ \mathit{st}) * \\
\quad \text{if lowest_closed } l = \text{Some } k \text{ then } \mathit{fd} = k \text{ else } 3 \le \mathit{fd} * \mathsf{ufd}\ \gamma\ \mathit{fd}\ \mathit{st}
\end{array}
$$

$$
\begin{array}{l}
m[\mathtt{a7}] = \mathtt{SYS_dup} \to m[\mathtt{a0}] = \mathit{fd}_0 \to \mathit{st} \neq \mathsf{Closed} \to \\
\mathsf{uinstr}\ \gamma\ \mathit{pc}\ \mathtt{ECALL} \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m\ \mathit{pc}\ n \mathrel{-\!\!*} \\
\mathsf{ustd}\ \gamma\ l \mathrel{-\!\!*} \mathsf{ufd_own}\ \gamma\ l\ \mathit{fd}_0\ \mathit{st} \mathrel{-\!\!*} \\
(\ \forall\ r. \\
\quad \big((\exists\ \mathit{fd}_1.\ \ r = \mathit{fd}_1 * \mathsf{ualloc}\ \gamma\ l\ \mathit{fd}_1\ \mathit{st} * \mathsf{ufd_own}\ \gamma\ (\mathsf{fill}\ l\ \mathit{st})\ \mathit{fd}_0\ \mathit{st}) \\
\quad \vee (r = -1 * \mathsf{ustd}\ \gamma\ l * \mathsf{ufd_own}\ \gamma\ l\ \mathit{fd}_0\ \mathit{st})\big) \mathrel{-\!\!*} \\
\quad \mathsf{urun}\ \gamma\ m[\mathtt{a0} := r]\ (\mathit{pc}+4)\ n \mathrel{-\!\!*} \mathsf{wp\ CpuLoop}\ ) \mathrel{-\!\!*} \\
\mathsf{wp\ CpuLoop}
\end{array}
$$

Figure 21: Descriptor resources and the rule for dup. The new descriptor's state is a copy of the source's. fill *l st* writes *st* into the lowest closed slot of *l*, if there is one.

This abstract view maps each live inode number to an anode: a file with its byte contents, a directory with its entries (including . and ..), or a device with its major and minor numbers, together with the link count. Unlinked inodes are not present in this abstract state, since their state can be lost on crash; the current specification does not support reasoning about programs that read/write unlinked-but-still-open inodes. This abstract state is not a tree because Unix allows hard links: specifying the file system as an abstract tree would require duplicating the state of each file at each of the file's linked names, and writing to a file would require finding and updating all of the links to that file in the tree.

$$
\begin{array}{l}
\mathsf{absnode} \triangleq \mathsf{File}\ (\mathit{bs} : \mathit{list\ byte}) \mid \mathsf{Dir}\ (\mathit{ents} : \mathit{fname} \rightharpoonup \mathit{inum}) \mid \mathsf{Dev}\ (\mathit{major\ minor} : \mathbb{Z}) \\
\mathsf{anode} \triangleq \{\mathit{node} : \mathsf{absnode};\ \mathit{nlink} : \mathit{nat}\} \\
\mathsf{aview} \triangleq \mathit{inum} \rightharpoonup \mathsf{anode}
\end{array}
$$

Figure 22: The abstract file system state.

Since the file system is shared between all of the processes running on top of the kernel, concurrent system calls might access or modify the same file system state. For example, one process might write data to the end of a log file, while another process reads the same log file.

To capture a concurrent specification for system calls that access the file system, the xv6 kernel provides atomic-update-style specifications. The atomic-update specification exposes the linearization point (or multiple linearization points, if necessary) of each system call, which defines the instant at which the effect of the system call can be thought of as taking place.

Specifying read. Figure 23 shows the specification for read on a file descriptor that refers to an inode, in the same style as the dup rule of Figure 21. The caller owns its descriptor resource (which says that *fd* is open for reading on inode *i*) and its destination buffer, and supplies a proof callback, read_commit, written using the $\Rrightarrow$ notation that §4.7 used for AMOSWAP. The kernel proof invokes the callback at the linearization point of the read, lending it the current abstract state of the file system *av*, the descriptor's current offset *off*, and the number of bytes *r* that the read is about to deliver. The callback must hand the file system state back unchanged, but it gets to learn facts about that state, captured by a predicate Φ of the application's choosing. For example, an application reading a record out of a database file can choose a Φ that snapshots the file's contents *bs* at the linearization point; the postcondition then ties the returned bytes to a slice of that same *bs*, so the application concludes that it read its record, and not an interleaving of two concurrent writers.

The offset is part of the callback because a file descriptor, and thus its offset, can be shared by multiple processes that read from it concurrently. A process that shares its descriptor hands the offset back unchanged, and the kernel advances it. A process that is the only user of a descriptor can instead own the other half of the offset ghost state, in which case its callback advances the offset by *r*; this allows the process to track its position in the file from one read to the next.

$$\mathsf{read_commit}\ i\ \gamma_o\ \Phi \triangleq \forall\ av\ off\ r.\ \ \mathsf{fs_auth}^{1/2}\ av \mathrel{-\!\!*} \mathsf{off}^{1/2}_{\gamma_o}\ off \Rrightarrow \mathsf{fs_auth}^{1/2}\ av * \mathsf{off_ret}\ \gamma_o\ off\ r * \Phi\ av\ off\ r$$
$$\mathsf{off_ret}\ \gamma_o\ off\ r \triangleq \mathsf{off}^{1/2}_{\gamma_o}\ off \vee \mathsf{off}^{1/2}_{\gamma_o}\ (off + r)$$

$$\begin{array}{l}
m[\mathtt{a7}] = \mathtt{SYS_read} \to m[\mathtt{a0}] = fd \to m[\mathtt{a1}] = dst \to m[\mathtt{a2}] = n \to 0 \le n \to \\
\mathsf{uinstr}\ \gamma\ pc\ \mathtt{ECALL} \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m\ pc\ k \mathrel{-\!\!*} \\
\mathsf{ufd}\ \gamma\ fd\ (\mathsf{Open\ true\ _\ (Inode}\ i\ \gamma_o)) \mathrel{-\!\!*} \mathop{\Large *}_{j<n} dst + j \mapsto_{\mathsf{uData}\,\gamma}\ f\ j \mathrel{-\!\!*} \mathsf{read_commit}\ i\ \gamma_o\ \Phi \mathrel{-\!\!*} \\
(\ \forall\ r\ g. \\
\quad \mathsf{ufd}\ \gamma\ fd\ (\mathsf{Open\ true\ _\ (Inode}\ i\ \gamma_o)) \mathrel{-\!\!*} \mathop{\Large *}_{j<n} dst + j \mapsto_{\mathsf{uData}\,\gamma}\ g\ j \mathrel{-\!\!*} \\
\quad \big(\ r = -1\ \vee\ \exists\ av\ off\ bs.\ \ av[i].node = \mathsf{File}\ bs \wedge r = \min(n, |bs| - off) \wedge \\
\qquad (\forall\ j < r.\ \ g\ j = bs[off + j]) \wedge (\forall\ j \ge r.\ \ g\ j = f\ j)\ *\ \Phi\ av\ off\ r\ \big) \mathrel{-\!\!*} \\
\quad \mathsf{urun}\ \gamma\ m[\mathtt{a0} := r]\ (pc + 4)\ k \mathrel{-\!\!*} \mathsf{wp\ CpuLoop}\ ) \mathrel{-\!\!*} \\
\mathsf{wp\ CpuLoop}
\end{array}$$

Figure 23: Atomic-update specification for `read` on an open, readable inode file descriptor. fs_auth is the abstract file system state of Figure 22, and off_{γ_o} is the offset of the file descriptor; the kernel lends its half of each to the callback.

When `read` returns, the caller gets its buffer back, with some prefix modified. On success, the return value is exactly $\min(n, |bs| - off)$, the rewritten prefix is the corresponding slice of the file, and the caller receives its Φ. On failure (e.g., the kernel could not copy the data into the user's buffer), the callback is either returned unused or exchanged for a zero-length receipt, which the figure elides.

The `read` rule has two more *arms*, not shown in Figure 23, that can be selected by the caller's knowledge of its descriptor's state: the console and a pipe end. They replace only the callback and the content clause. For example, for a console file descriptor, the callback is an atomic update against the console's *input history* (the kernel's read event, written as a $\Rrightarrow$ callback) and the content clause says the delivered bytes are the next segment of that history.

Specifying `open`. The specification of `open` must deal with the fact that resolving a path name is not atomic: the kernel walks the path one directory at a time, locking each directory only while it looks up the next path element, so other processes can concurrently create, link, and unlink entries along the path. As a result, `open` has not one linearization point but one per path element, plus a final one at the inode being opened. Figure 24 shows the resulting specification, for `open` without the `O_CREATE` and `O_TRUNC` flags.

$$\begin{array}{l}
\mathsf{hop}\ P\ P_{miss}\ k\ s \triangleq \forall\ d\ ents.\ \ P\ k\ d \mathrel{-\!\!*} \mathsf{dir}\ d\ ents \Rrightarrow \mathsf{dir}\ d\ ents * (\mathsf{if}\ ents[s] = \mathsf{Some}\ d'\ \mathsf{then}\ P\ (k+1)\ d'\ \mathsf{else}\ P_{miss}\ k\ d) \\
\mathsf{walk}\ c\ P\ P_{miss}\ path \triangleq (\mathsf{True} \Rrightarrow P\ 0\ (\mathsf{if}\ path[0] = \texttt{'/'}\ \mathsf{then\ ROOT\ else}\ c)) * \mathop{\Large *}_{k<|\mathsf{elems}\ path|} \mathsf{hop}\ P\ P_{miss}\ k\ (\mathsf{elems}\ path)[k] \\
\mathsf{open_commit}\ \Phi \triangleq \forall\ av\ i.\ \ \mathsf{fs_auth}^{1/2}\ av \Rrightarrow \mathsf{fs_auth}^{1/2}\ av * \Phi\ av\ i
\end{array}$$

$$\begin{array}{l}
m[\mathtt{a7}] = \mathtt{SYS_open} \to m[\mathtt{a0}] = p \to m[\mathtt{a1}] = omode \to \\
\mathsf{uinstr}\ \gamma\ pc\ \mathtt{ECALL} \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m\ pc\ n \mathrel{-\!\!*} \mathsf{ustr}\ \gamma\ p\ path \mathrel{-\!\!*} \mathsf{ucwd}\ \gamma\ c \mathrel{-\!\!*} \mathsf{ustd}\ \gamma\ l \mathrel{-\!\!*} \\
\mathsf{walk}\ c\ P\ P_{miss}\ path \mathrel{-\!\!*} \mathsf{open_commit}\ \Phi \mathrel{-\!\!*} \\
(\ \forall\ r. \\
\quad \big(\ (r = -1 * \mathsf{ustd}\ \gamma\ l * \ldots) \vee \exists\ fd\ av\ i.\ \ r = fd * P\ |\mathsf{elems}\ path|\ i * \Phi\ av\ i * \\
\qquad \mathsf{ualloc}\ \gamma\ l\ fd\ (\mathsf{Open\ (readable}\ omode)\ (\mathsf{writable}\ omode)\ (\mathsf{type_of}\ i\ av[i].node))\ \big) \mathrel{-\!\!*} \\
\quad \mathsf{ustr}\ \gamma\ p\ path \mathrel{-\!\!*} \mathsf{ucwd}\ \gamma\ c \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m[\mathtt{a0} := r]\ (pc + 4)\ n \mathrel{-\!\!*} \mathsf{wp\ CpuLoop}\ ) \mathrel{-\!\!*} \\
\mathsf{wp\ CpuLoop}
\end{array}$$

Figure 24: Specification for `open` of an existing file. dir d *ents* is the part of the abstract file system state (Figure 22) saying that inode d is a directory with entries *ents*; elems splits a path into its elements; ustr and ucwd are the caller's path string and current directory; and ualloc is from Figure 21.

The caller supplies a proof callback for every *hop* of the walk. The callbacks are tied together by a *cursor* predicate P chosen by the caller, where $P\ k\ d$ means that, after k hops, the walk has reached the inode d. At hop k, the kernel lends the callback the entries *ents* of the directory that the walk is currently in; the callback hands them back unchanged, and exchanges its cursor $P\ k\ d$ for $P\ (k+1)\ d'$ if the directory maps the path element s to inode d', or for $P_{miss}\ k\ d$ if there is no such entry, in which case `open` will fail. The walk starts at the root directory for an absolute path, and at the caller's current directory otherwise. Once the walk reaches the final inode, the kernel proof invokes open_commit, which works like the callback for `read`: it lets the caller learn facts Φ about the file being opened, at

the instant it is opened. The callback is quantified over the inode i; the postcondition delivers $\Phi\ av\ i$ at the same i as the final cursor, which is how the caller relates the two.

On success, the caller gets back the final cursor, its Φ, and a new file descriptor at the lowest closed slot, as with `dup`; the descriptor's type is determined by the inode that the walk reached, so that, for example, a program that opens `/console` learns that its descriptor refers to the console device. On failure, the caller gets back whatever cursor the walk ended with, along with its unused callbacks, which the figure elides.

The caller's choice of P determines what it learns from the walk. A caller that knows nothing about the file system can choose a trivial P, and learns only that `open` returned some descriptor or failed. A caller that holds ghost state pinning down the directories along its path—as our top-level theorem application does for the root directory, which contains `/init`, `/sh`, `/echo`, etc.—can choose a P that tracks the exact inode the walk has reached, and that makes P_{miss} contradictory; such a caller learns precisely which inode it opened. The same path-walk callbacks appear in the specification of every system call that takes a path name, including `exec`.

7.4 Processes

The main system calls in xv6 used to manage processes are `fork`, `exit`, `wait`, and `exec`. The specifications for `fork`, `exit`, and `wait`, as provided by the user-level proof library, are shown in Figure 25.

$$
\begin{array}{l}
\mathsf{Forkable}\ P \to m[\texttt{a7}] = \texttt{SYS_fork} \to \\
\mathsf{uinstr}\ \gamma\ pc\ \texttt{ECALL} \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m\ pc\ n \mathrel{-\!\!*} P\ \gamma \mathrel{-\!\!*} \mathsf{ufds}\ \gamma\ D \mathrel{-\!\!*} \mathsf{uch}\ \gamma\ S \mathrel{-\!\!*} R \mathrel{-\!\!*} \Box(T \mathrel{-\!\!*} Q\ (-1)) \mathrel{-\!\!*} \\
(\ \forall\ r.\ \ r \neq 0 \to \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(parent)} \\
\quad \big((r = -1 * \mathsf{uch}\ \gamma\ S * R) \ \vee\ \exists\ \gamma_c.\ \ \mathsf{child_tok}\ \gamma_c\ r\ Q * \mathsf{uch}\ \gamma\ (S \cup \{\gamma_c\})\big) \mathrel{-\!\!*} \\
\quad P\ \gamma \mathrel{-\!\!*} \mathsf{ufds}\ \gamma\ D \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m[\texttt{a0} := r]\ (pc+4)\ n \mathrel{-\!\!*} \mathsf{wp\ CpuLoop}\ ) * \\
(\ \forall\ \gamma'.\ \ \gamma'.\mathit{exit} = Q \to \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(child)} \\
\quad P\ \gamma' \mathrel{-\!\!*} \mathsf{ufds}\ \gamma'\ D \mathrel{-\!\!*} \mathsf{uch}\ \gamma'\ \emptyset \mathrel{-\!\!*} R \mathrel{-\!\!*} \mathsf{urun}\ \gamma'\ m[\texttt{a0} := 0]\ (pc+4)\ n \mathrel{-\!\!*} \mathsf{wp\ CpuLoop}\ ) \mathrel{-\!\!*} \\
\mathsf{wp\ CpuLoop}
\end{array}
$$

$$
m[\texttt{a7}] = \texttt{SYS_exit} \to \mathsf{uinstr}\ \gamma\ pc\ \texttt{ECALL} \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m\ pc\ n \mathrel{-\!\!*} \gamma.\mathit{exit}\ m[\texttt{a0}] \mathrel{-\!\!*} \mathsf{wp\ CpuLoop}
$$

$$
\begin{array}{l}
m[\texttt{a7}] = \texttt{SYS_wait} \to \mathsf{uinstr}\ \gamma\ pc\ \texttt{ECALL} \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m\ pc\ n \mathrel{-\!\!*} \mathsf{uch}\ \gamma\ S \mathrel{-\!\!*} \\
(\ \forall\ r\ S'.\ \ \big((r = -1 \wedge S' = S)\ \vee\ \exists\ \gamma_c \in S.\ \exists\ xs.\ \ S' = S \setminus \{\gamma_c\} * \mathsf{exit_tok}\ \gamma_c\ r\ xs\big) \mathrel{-\!\!*} \\
\quad \mathsf{uch}\ \gamma\ S' \mathrel{-\!\!*} \mathsf{urun}\ \gamma\ m[\texttt{a0} := r]\ (pc+4)\ n \mathrel{-\!\!*} \mathsf{wp\ CpuLoop}\ ) \mathrel{-\!\!*} \mathsf{wp\ CpuLoop}
\end{array}
$$

$$
\mathsf{child_tok}\ \gamma_c\ pid\ Q \mathrel{-\!\!*} \mathsf{exit_tok}\ \gamma_c\ pid\ xs \mathrel{-\!\!*} \triangleright\ Q\ xs
$$

Figure 25: Specifications for `fork`, `exit`, and `wait`, and the rule for redeeming a child's exit resources. ufds $\gamma\ D$ stands for the caller's file descriptor resources (Figure 21) and its current directory; uch $\gamma\ S$ says that the process's un-waited-for children are S; and T is the application's taint (Figure 28).

Specifying `fork`. `fork`'s specification captures the fact that, when `fork` succeeds, it returns twice. The specification represents this fact by requiring the caller to supply two continuations, in case `fork` succeeds. One continuation, representing the parent's execution, continues running with the same γ that invoked `fork` in the first place, and the other continuation, representing the child's execution, runs with a new γ' representing the child's copy of the parent's process, including the memory address space, the register set, the file descriptors, and the program break. Both continuations receive the caller's file descriptor resources ufds, each at its own γ, because the child's descriptor table is a copy of the parent's. The parent additionally chooses a resource R to hand over to the child outright (e.g., `init` hands the shell the right to read console input); if `fork` fails, the parent gets R back.

One challenge with `fork` lies in representing its effects in separation logic. For example, consider the shell, which uses `fork` followed by `exec` to start a new process. The parent process has a memory buffer storing the command to be executed. After `fork` returns to the parent, the parent process may want to recycle that buffer. On the other hand, the proof of the child process needs to know the contents of that buffer to reason about what happens when it calls `exec`. Separation logic does not allow the same resource to be owned concurrently by both the parent and the child, and indeed the parent and child have separate copies of their memory after `fork` succeeds.

To allow the proof to learn that the initial state of the child process matches the state of the parent process at the time it called `fork`, the user-level proof library uses a notion of a Forkable predicate. A predicate P that depends on the γ of the current process is forkable if it can be represented in terms of a set of memory locations whose contents are preserved across `fork`. This means that, if $P\ \gamma$ holds before calling `fork` in the parent process, then $P\ \gamma$ still holds

in the parent after `fork` returns, and $P\ \gamma'$ holds in the child after `fork` returns there. The definition of Forkable is shown in Figure 26. For example, when the proof of the shell invokes the specification of `fork`, it might choose a predicate P that covers the `argv` buffers that are about to be passed to `exec`.

$$\begin{aligned}
&\mathsf{Forkable}\ P \triangleq \forall\ \gamma.\ \ P\ \gamma \mathrel{-\!\!*} \exists\ F_t\ F_p\ F.\\
&\quad \mathop{*}_{a \mapsto b \in F_t} a \mapsto_{\mathsf{uText}\ \gamma}\ b\ *\ \mathop{*}_{a \mapsto b \in F_p} a \mapsto^{\mathsf{ro}}_{\mathsf{uData}\ \gamma}\ b\ *\ \mathop{*}_{a \mapsto b \in F} a \mapsto_{\mathsf{uData}\ \gamma}\ b\ *\\
&\quad \Big(\mathop{*}_{a \mapsto b \in F} a \mapsto_{\mathsf{uData}\ \gamma}\ b \mathrel{-\!\!*} P\ \gamma\Big)\ *\\
&\quad \Box\Big(\forall\ \gamma'.\ \ \mathop{*}_{a \mapsto b \in F_t} a \mapsto_{\mathsf{uText}\ \gamma'}\ b \mathrel{-\!\!*} \mathop{*}_{a \mapsto b \in F_p} a \mapsto^{\mathsf{ro}}_{\mathsf{uData}\ \gamma'}\ b \mathrel{-\!\!*} \mathop{*}_{a \mapsto b \in F} a \mapsto_{\mathsf{uData}\ \gamma'}\ b \Rrightarrow P\ \gamma'\Big)
\end{aligned}$$

Figure 26: The Forkable class: a payload P over the heap's ghost names factors through a footprint of text, read-only data, and exclusive data bytes, and can be rebuilt at any fresh names γ'.

Specifying `exit` and `wait`. In separation logic, a running user-level process might own some exclusive resources. For example, the user-level shell spawned by `init` might own the exclusive resource for reading from the UART console input; this proves that there are no concurrent processes also reading from the console, and therefore the user's input is not misinterpreted by the shell. This resource is initially passed to the shell by `init`'s proof, when it invokes `fork`; the proof of `init` gets to choose which of its resources it keeps for itself in the parent process, and which of its resources it passes to the child.

However, when a child exits, these resources may need to be returned to the parent. In the case of xv6, when `init` observes that the shell exits, it will spawn another shell; to do this, the proof needs to obtain ownership of the UART console input back from the old shell, in order to give it to the new shell.

To represent this resource transfer, the specification of processes in xv6 associates an exit predicate, $\gamma.exit$, with each process. This predicate defines what resources must be returned by the process when it exits. This predicate is chosen in the parent's proof at the time it invokes `fork`, and the child process is responsible for ensuring that it returns these resources whenever it exits, as shown in the precondition of `exit` in Figure 25. Since `exit` terminates the process and never returns, its specification has no continuation.

A process may also be terminated by the kernel, with exit status -1, without ever calling `exit`: because of an exception (such as an illegal memory access), or because another process killed it using `kill`. The child cannot be expected to pay its exit resources in this case, since it can be terminated at any instruction. Instead, the specification makes $\gamma.exit$ a function of the exit status, and `fork` requires the parent to show that the resources for status -1 follow from the application's *taint* T, which we describe in §8 (Figure 28). Informally, T holds only once the application's top-level claim has been abandoned, at which point nothing more needs to be proven about the exit resources. This corresponds to the "unspecified output" case of Figure 2, which captures the case of a user typing in `echo xx > /sh`. Verified programs, in turn, prove that they trigger no exceptions, and that they do not kill processes that do not expect it.

Finally, when the parent calls `wait`, it obtains the resources associated with the exit predicate of the now-dead child process. When the process calls `wait`, it does not know yet which PID will be reported as exited. The specification deals with this by having `wait` return an exit token, exit_tok, for a particular child generation and PID. The proof of the caller (e.g., `init`) can combine this exit token together with its knowledge of what the exit predicate of the child process is, child_tok, to obtain ownership of the child's exit resources Q, using the last rule in Figure 25.

One complication is that Unix allows PIDs to be reused over time, so a PID alone does not identify a child. The specification instead identifies each incarnation of a process by a ghost name, its *generation*: `fork` returns the child's generation γ_c in child_tok and adds it to the parent's set of children S, and `wait` reports which generation it reaped, which is what lets the parent match the exit token against the right child_tok.

Specifying `exec`. The `exec` system call introduces a new complication: if the system call succeeds, then it returns to a whole new process image, as defined by the ELF binary that was executed, and starts executing at the entry point of that ELF binary. Figure 27 shows how the xv6 specification for `exec` captures this fact. In particular, there are two continuations needed by the `exec` specification. If `exec` errors out, it returns -1 to the caller as usual, and the caller must supply a continuation for resuming at the next instruction with the return value being -1; this is the explicit continuation in the specification. The continuation for the success case is part of the *exec deposit*, exec_deposit, which bundles the one-shot resources that the caller hands to the kernel: the path-walk callbacks for resolving the path name of the binary (the same walk as for `open` in Figure 24); the same open_commit callback, for the kernel's single

observation of the file's contents, which it makes while holding the inode's lock; and the *slot* premise exec_slot_pre, which holds the success continuation.

$$m[\texttt{a7}] = \texttt{SYS_exec} \to \text{uinstr } \gamma\ pc\ \texttt{ECALL} \mathrel{-\!\!*} \text{urun } \gamma\ m\ pc\ n \mathrel{-\!\!*} \text{exec_deposit } \gamma\ m\ pc \mathrel{-\!\!*}$$
$$(\ \text{urun } \gamma\ m[\texttt{a0} := -1]\ (pc+4)\ n \mathrel{-\!\!*} \text{wp CpuLoop}\ ) \mathrel{-\!\!*} \text{wp CpuLoop}$$

$$\text{exec_deposit } \gamma\ m\ pc \triangleq \text{walk } c\ P\ P_{miss}\ path * \text{open_commit } \Phi * \text{exec_slot_pre } Q\ na\ argv\ fds * \ldots$$

$$\text{exec_slot_pre } Q\ na\ argv\ fds \triangleq \forall\ f\ W'.\ \text{observed } f \mathrel{-\!\!*} \text{loadable } f \to$$
$$\text{image_ok } f\ na\ argv\ fds\ W' \to \text{my_pay } W'.gen\ Q \mathrel{-\!\!*} \text{uslot } W'$$

$$\text{uslot } W \triangleq \forall\ pt\ C.\ \text{ukernel } pt\ C\ W \mathrel{-\!\!*} \text{wp CpuLoop}$$

$$\text{image_ok } f\ na\ argv\ fds\ W' \triangleq$$
$$W'.tf.\texttt{pc} = \text{entry } f \wedge W'.sz = \text{top } f \wedge W'.tf.\texttt{sp} = W'.tf.\texttt{a1} = \text{sp_final } f\ argv \wedge W'.tf.\texttt{a0} = na \wedge$$
$$\text{elf_image } f \subseteq W'.M \wedge \text{args_at (top } f)\ na\ argv\ W'.M \wedge \text{stack_zero (top } f)\ argv\ W'.M \wedge$$
$$\text{perm_ok } f\ W'.\pi \wedge W'.fds = fds$$

Figure 27: Simplified specification for exec. The success case is the slot uslot W' inside exec_slot_pre, which is in turn part of exec_deposit; ukernel is from Figure 19; walk and open_commit are from Figure 24, and observed f is what they deliver when the path resolves to a file with contents f.

On the other hand, if exec succeeds, the caller is responsible for supplying a continuation for running in a whole new process state, as defined by uvis: for every file f that the kernel's observation may deliver, and every process state W' that the kernel may build from it, the slot premise requires the caller to supply uslot W': a wp CpuLoop for running the new program, given the kernel's resources for a user process whose visible state is W' (the same ukernel that underlies urun in Figure 19). The image_ok predicate defines the correspondence between the contents of the ELF binary that was passed to exec and the new state of the process (the initial memory contents, register values, etc.). The file descriptors in the new process image are inherited from the previous process image. To reason about the contents of the ELF file, the exec specification uses an atomic-update-style pattern much like other file system syscalls that we described earlier.

8 Specifying user-level programs and shell commands

The previous section describes the specifications of system calls for xv6, which allow proving theorems about user-level applications such as echo, cat, grep, etc. However, we would like to reason about console-level behavior of a user interacting with a shell, where the user can invoke combinations of these commands, such as `cat README | grep xv6 > out.txt`.

Abstract specs for shell programs. To avoid having to prove every possible shell pipeline separately, we write an abstract specification capturing each program's behavior in terms of what it does to its file descriptors. In particular, we write an abstract model of the program's behavior in terms of an interaction tree (ITree) [104], and prove that the implementation matches this specification. This allows us to abstract over the different choices for what stdin and stdout might be at runtime, such as the input or output of a given program being the console, a file, or a pipe.

Sandboxing with `seccomp`. To demonstrate that xv6 can provably isolate arbitrary user-level code, we implemented a simple sandboxing mechanism in xv6, seccomp, mirroring the Linux mechanism with the same name. This seccomp mechanism allows specifying a set of syscalls that are blocked, and this blocked set is inherited by all child processes. We implement a simple wrapper command that allows a user to run arbitrary binaries under this sandbox; the wrapper blocks the open, link, unlink, mkdir, mknod, and kill system calls. It allows other syscalls, including read and write (so that the program can interact with open file descriptors inherited from its parent), memory allocation, fork and exec, etc. The blocked system calls, however, ensure that the sandboxed program cannot modify the contents of the file system (except through any inherited file descriptors), or affect other processes running on the kernel.

Application is an invariant. To prove a theorem about the behavior of shell commands, we need to give ownership of different parts of the system (such as exclusive ownership of console input and output, or knowledge of the contents of the file system) to different processes over time. For instance, when init calls exec on /sh, its proof needs to know that the file /sh contains the shell's (verified) binary, even though any process could modify that file; and when echo

writes its arguments to the console, its proof needs to know that nothing else is writing to the console at the same time.

MachCSL captures these facts as an *application invariant*: a CSL invariant, chosen by the application, that holds in addition to the kernel's own invariants. Figure 28 sketches the invariant for the top-level theorem's application. It says that the binaries of `/init`, `/sh`, `/echo`, etc., in the file system are the ones from the initial disk image (pinned), and that the bytes sent to the console so far form a legal transcript for the input received so far (legal), in the sense of Figure 2.

The application invariant is tied to the kernel's state by splitting ownership of that state: the application invariant holds one half of the authoritative ghost state for the abstract file system and for the console's history, and the kernel's invariants hold the other half. The two halves always agree, and updating the state requires both. As a result, the kernel cannot change the file system or send a byte to the console without the application invariant being re-established for the new state, even though nothing in the kernel's own invariants mentions the application.

Processes maintain the invariant at their system calls. The kernel does not know how to re-establish an application's invariant, so this is the job of the process that issued the system call. This is the role of the proof callbacks ($\Rrightarrow$) in the system call specifications, such as read_commit in Figure 23 and the hops of walk in Figure 24: at the system call's linearization point, the kernel proof invokes the caller's callback, and the callback opens the application invariant, combines its half of the state with the half lent by the kernel's proof, updates the state, and re-establishes the invariant. For example, the callback that `echo`'s proof passes to `write` must show that appending `echo`'s output to the console history keeps the transcript legal.

To prove this, `echo` needs to know that it is its turn to write, and what it is supposed to print. This knowledge takes the form of resources that flow between the process proofs along with control: `init` passes them to the shell through `fork`, the shell passes them to `echo`, and `wait` returns them through the exit predicate (Figure 25). The same callbacks let a process *use* the invariant: the callbacks that `init` passes to `exec` for `/sh` open the invariant to learn that the path resolves to the shell's binary, which allows `init`'s proof to continue with the proof of the shell as its continuation. Each process proof is thus a proof about one binary, and the application invariant is what connects them.

The invariant implies the trace property. The kernel's top-level theorem is parameterized by the application. To instantiate it, the application supplies three things: a proof that its invariant holds for the initial disk image (a computation over the bytes of `fs.img`) and is re-established at every boot; a proof of the first user process, `init`, against the system call specifications, which in turn needs proofs of the programs that it forks and execs; and a proof that the invariant implies the desired property of the externally visible trace. The last step is simple for our top-level theorem's application, because the invariant directly states that the console output is a legal transcript for the console input. The resulting theorem is the one from §2: every execution of the machine is safe, and its trace is one of the traces allowed by Figure 2.

Taint. The application invariant cannot hold unconditionally, because the user at the console may type anything. If the user types `echo xx > /sh`, as in Figure 2(d), the shell will overwrite its own binary, and pinned no longer holds. The kernel must remain correct when this happens (i.e., the user should not be able to corrupt the kernel), and indeed the theorem still promises that the kernel maintains all of its own invariants; what is lost is only the application's promise about the console output.

MachCSL represents this using a *taint* T: a persistent fact saying that the console input has departed from the discipline that the application expects. The application states every one of its claims as a *lease*, which is either the claim or the taint. The invariant tracks whether the input so far is disciplined using a monotone counter, which goes from 0 to 1 when the input first leaves the discipline, and can never go back; T is the knowledge that the counter has reached 1. Process proofs case-split on each lease they hold. In the untainted case, they have the claim, and proceed as described above. In the tainted case, they hold T, which trivially re-establishes every lease in the invariant, so a tainted process has no further application-level obligations; its proof falls back to the kernel's generic guarantee for arbitrary user code (§6.13), which is also what covers any unverified binary that a tainted system might run. The taint also covers the kernel terminating a process, which is why it appears in the specification of `fork` (Figure 25). Finally, since T contradicts the counter being 0, the invariant implies that a disciplined input has a legal transcript, which is exactly the shape of the theorem: the trace property is promised only for inputs that follow the discipline, and kernel safety is promised always.

MachCSL also maintains a per-era taint resource that represents the fact that the user ran `seccomp xx` for some untrusted binary `xx`. The reason that this taint is necessary is that, once a user runs `xx` under the seccomp sandbox, that binary could keep a copy of the console file descriptor until the system reboots. As a result, any input or output of the system is now suspect; e.g., even if the user sees what appears to be a legitimate shell prompt, it might actually be a fake shell prompt printed by the `xx` program. Thus, the top-level theorem says that, after a user runs `seccomp xx`, the output could be anything, until the machine reboots, at which point it becomes well-defined again.

$$T \triangleq \mathsf{mono_lb}\ \gamma_T\ 1 \qquad\qquad \mathsf{lease}\ R \triangleq R \lor T$$

$$\mathsf{app_inv} \triangleq \boxed{\begin{array}{l}\exists\, av\, h.\ \ \mathsf{fs_auth}^{1/2}\ av * \mathsf{cons_auth}^{1/2}\ h * \mathsf{mono_auth}\ \gamma_T\ (\text{if disc } h \text{ then } 0 \text{ else } 1) * \\ \quad \mathsf{lease}\ \lceil \mathsf{pinned}_{\texttt{/init,/sh,/echo}}\ av \rceil * \mathsf{lease}\ \lceil \mathsf{legal}\ h \rceil\end{array}}$$

$$\mathsf{app_inv} * \mathsf{cons_auth}^{1/2}\ h \vdash \lceil \mathsf{disc}\ h \to \mathsf{legal}\ h \rceil$$

Figure 28: A sketch of the top-level theorem's application invariant. av is the abstract state of the file system (Figure 22) and h is the history of console input and output; the kernel's invariants hold the other halves of fs_auth and cons_auth. disc and legal are the allowed inputs and the legal transcripts of §2 (Figure 2). The last line is how the invariant implies the trace property.

9 Validating the model

To gain confidence in the trusted RISC-V model that underlies MachCSL, we developed a set of conformance tests that cross-check the model against a reference implementation. In our case, we use three reference implementations: the QEMU rv64 whole-system emulator, whose `virt` machine is the hardware platform targeted by xv6; the StarFive VisionFive 2 RISC-V board (JH7110 SoC), manipulated via JTAG; and the CVA6 implementation of a RISC-V system, simulated in Verilator. The goal of the conformance checker is to check that MachCSL's formal model captures the possible behaviors of the reference implementations.

Formally stated, the conformance checker's job is to find executions of the reference implementation and prove that the observed executions are a subset of the executions allowed by the model as defined in Rocq. This gives us confidence that our theorem, which applies to every execution of the xv6 kernel on top of the trusted model, covers the executions on top of the reference implementation as well.

One subtle aspect of the correspondence between the reference implementation and the model is that, in some cases, the model gets "stuck", meaning that the semantics of a certain operation is undefined. For example, the MachCSL model of the disk does not support certain request types, which are not used by xv6; the semantics is undefined for those operations. The MachCSL theorems prove that the xv6 kernel never triggers this undefined behavior. Thus, for the purpose of the conformance checker, a test case on which the model gets stuck (i.e., has no transition at all) counts as passing: it is treated as a valid correspondence between the reference implementation and the model.

Conformance checking is purely a property of the model itself, and does not involve any of the CSL machinery used to prove correctness of the concurrently executing xv6 kernel. This makes the conformance checker's job easier: in many ways, it suffices to simply let the model execute for many cycles, compute the result of that execution, and check whether it matches the reference output.

There are two key challenges in conformance checking. The first is to generate enough test cases to cover all of the interesting aspects of the system's execution. The second is to deal with non-determinism, which has two facets. On the reference implementation side, if some test case produces multiple different outputs, due to some non-determinism, it is the job of the conformance checker to generate as many of those outputs as possible. On the model (Rocq) side, proving that a certain output can be observed in the model may require a specific choice of non-determinism, such as uninitialized initial register contents or interleaving between concurrent CPUs and devices; it is then the conformance proof's job to synthesize this schedule of non-deterministic events to demonstrate that the reference output can be reproduced by the model.

We implement the test cases as short snippets of RISC-V assembly instructions. The output of a test case is the contents of a 4 KB page of memory; the particular test case chooses what relevant data it wants to include in that 4 KB page. The conformance checker must prove that every 4 KB page snapshot observed in the reference implementation (QEMU, JH7110, or CVA6) can also be reproduced on top of the model.

Our conformance tests include tests for basic CPU operation: simple instructions, contents of the registers at system boot time, and page-table and interrupt machinery. These tests are largely deterministic.

The conformance tests also include tests for shared-memory behavior. In order to exhibit memory access races, the test first runs a setup phase, which waits for all of the CPUs to boot, followed by a single barrier that releases all of the CPUs into the second phase, which tries to execute a racing shared-memory access. This generates multiple possible results (4 KB page snapshots) for a single test, reflecting multiple possible race outcomes. On the model side, for each observed result, the proof constructs an explicit schedule of steps that leads to it.

The tests also include test cases for devices, including the UART, the PLIC, and the virtio disk, as well as the disk's DMA machinery (the UART and disk tests run only on QEMU, since the JH7110 and CVA6 have a different UART and no virtio disk). The RISC-V assembly instructions directly access the device MMIO registers, set up shared DMA ring buffers, etc.

10 Agent-based verification

At a high level, using agents for this verification project looks just like writing code with an agent. We have many Claude Code agents running in parallel, each with its own directory that contains a checkout of the project's git repository. The agent changes files, runs `make` to check whether the proofs are correct, modifies specs, proofs, and any other files in the directory, commits, pushes, resolves merge conflicts, etc. The repository contains Rocq files; `make` runs Rocq to check that the proofs are all valid, and we have a CI job that continuously makes sure that the proofs check out (and reports the axioms that the top-level theorem depends on).

As with other agent-based software development efforts, we find it is useful to keep design notes, optimization techniques, ongoing and completed projects, failed approaches that should not be tried again, etc., as documentation (in Markdown format) in the git repository itself. This keeps the agents' memory in the git repository, shared between all of the agents working on the project, as opposed to in per-agent local files on the agent's machine.

The project's proof development is large (1.6 million lines of Rocq), but most of it consists of proofs, which are, at some level, irrelevant as long as they are validated by Rocq and prove the desired top-level theorem. The one caveat is performance: we continuously keep an eye on proof-checking times because proofs that take a long time to check directly impact the iteration cycle needed to make further progress, as each rebuild to check proof validity gets more expensive.

Even the specifications are, for the most part, not fully trusted. They are important to get right, in order to ensure that different parts of the proof will compose correctly with one another. However, the top-level adequacy theorem cancels out all of the intermediate specs and proves a much simpler theorem. This ensures that, if the agents do not get the individual function specs quite right, in a way that creates conflicts between different kernel subsystems, this will be caught because, when `make` runs Rocq to check the top-level adequacy theorem, Rocq will reject the proof. This also allows us to be less diligent about auditing the intermediate specs, as long as the top-level theorem can still be proven.

10.1 Abstractions

Agents, such as Claude Code, are effective at automating proofs, but one thing that we found agents are not at all good at is introducing appropriate abstractions. For example, when we were verifying the file system, the agent's specifications and proofs about all file system functions involved direct reasoning about the contents of on-disk blocks. As a result, the specifications were extremely verbose and fragile. We forced the agent to introduce abstract state capturing the notion of an inode, a directory, and a file system tree. This greatly simplified the specifications, made them more robust to low-level changes, and allowed the agent to finish the proofs, whereas before this abstraction, it could not scale to the level of complexity generated by the explicit statements about all on-disk bits. We had many other examples of this failure mode, including abstractions for reasoning about the kernel's instructions living in physical memory, abstractions for page-table translation and TLB consistency, abstractions for a kernel thread running with its own kernel stack, abstractions for ABI calling conventions, abstractions for interrupt handling, etc.

Another weakness of current agents is that they are not good at reasoning about concurrency. In particular, setting up ghost state, invariants, and ownership disciplines in terms of which functions own which separation-logic facts requires manual effort. On the other hand, once a function has a clear separation-logic spec, and clear invariants owning separation-logic facts about shared resources, agents are effective at doing the actual proofs.

10.2 Iterating on the design

Several times, we asked an agent to make a substantial change to the specs and proofs, and the agent was ultimately unable to complete the task. The failure mode, in each case, was that the agent entered an infinite spiral, continuing to introduce more lemmas, theorems, specs, case analyses, etc., without really making progress on the overall problem. For example, when we asked the agent to prove the safety of executing user-mode code, it started formulating

abstractions and theorems about all the possible instructions that a user process might execute, all the possible memory contents that the process might have, etc., all without a clear end in sight. When we asked the agent to add the notion of a current HART ID to every WP, to allow talking about process migration across cores, the agent devolved into rewriting every specification and proof, without converging. When we asked the agent to prove `ialloc`, it went as far as modifying the xv6 code and adding a new axiom to paper over a bug (that it did not spot), rather than stepping back. When we asked an agent to introduce a layer of abstraction for virtual memory translation backed by the kernel's real page table (as opposed to the identity mapping), the agent started extending every single specification to talk about the bit-level contents of page-table entries corresponding to every address mentioned in that spec.

A final example of this kind of failure showed up when we were proving file system consistency across reboots. Agents initially came up with internal invariants that were sufficient to prove kernel safety (bounds on link counts, etc.), but not sufficient to connect on-disk block-level state to abstract state like files and directories. As a result, all of the specs leaked the low-level byte details, and then committing a transaction required reasoning about the byte-level effects of the transaction's writes, and whether the file system invariant would still hold if log recovery ran after a crash with those byte-level writes. Every syscall had to prove that, if log recovery ran on its changes, the effect would still satisfy the global FS consistency invariant. We had to intervene and force two changes to the specs: (1) a hard layer boundary at the write-ahead log, proving an atomic update from old logical disk state to new logical disk state, and avoiding any reasoning about individual writes or log recovery, and (2) invariants that connect block-level state to abstract file/directory state, rather than invariants that define inequalities to avoid panics.

The common theme in all of these failure modes is that we did not fully understand the problem that we were asking the agent to solve. In each case, after observing the agent entering this spiral for several days, we better understood the problem space, including the challenges that we had to address in formulating our goal for the agent. The solution in each case turned out to be stopping the agent, discarding its work, and starting over with a more precise goal statement for the agent, informed by the previous agent's failures. In three cases, it took us three iterations to finally reach a workable solution (the kernel page-table support, the user-space execution theorem, and file system crash consistency), because these cases turned out to require a lot of attention to specific details that were not obvious to us from the outset.

We also found it helpful to use agents to help us fine-tune specifications. When we had a rough specification that we were not sure about, we asked the agent to go ahead and try to prove it, knowing full well that it was likely to fail. However, since the cost of synthesizing the proof using an agent is relatively low, this was a highly effective way for us to learn where the remaining tricky cases were in our spec; having a fully worked-out proof that fails at exactly those tricky cases was helpful feedback for us to improve the spec.

10.3 Multi-agent design review

In several cases, we found it effective to ask the agent to write a design document describing its proposed plan for a complex change, and have other agents explicitly review that design, cross-reference it with the actual code, and sketch out the core proof argument in Rocq to get the syntax correct and type-checked (without fully proving it).

This came up when an agent was porting particularly tricky proofs (the buffer cache and inode cache) from a sequentially consistent memory model to the TSO weak-memory model. The agent was in an endless spiral for about two days, continuously finding problems with the design, revising the design, finding more problems, etc. Several times, we asked the agent to take a step back and re-think the design, but that did not help.

What turned out to work well was to ask a second agent to take a look at all of the design documents written by the first agent, analyze its multiple attempts at solving the problem, and come up with a plan to avoid the pitfalls that led to the first agent's spiral. This quickly surfaced a number of deep issues that made the first agent's design unworkable. A third agent, acting as a second reviewer, spotted further issues that the first reviewer missed on its own. We handed this new design document back and forth between the implementation agent and the two reviewers about ten times, in the span of a few hours; each iteration spotted fresh issues that none of the agents found by themselves. Along with the design document, we asked the agents to develop (and iterate on) a prototype of the overall proof plan, in Rocq, containing the relevant definitions, specifications, and invariants, with the actual proof part assumed for the time being. Afterwards, the implementation agent completed the task (finishing the buffer-cache proofs) in a single pass of about two hours. A similar approach worked for completing the inode-cache proofs, albeit with more review round-trips, owing to the higher complexity of the inode cache.

10.4 Iterative discovery of specifications is slow

The part of the overall proof development process that takes the longest is making changes to existing specifications—e.g., strengthening an existing syscall specification to promise more facts, changing a kernel invariant to track more information about the state of a process, etc. These changes take a long time because they often require modifying many existing files that touch the specs or invariants that are being updated.

As a result, one style of proof development that is particularly slow is iterative specification discovery, where the complete specification or invariant is not precisely known ahead of time. In this setting, the proof developer, together with the agent, repeatedly tries to prove some theorem, fails because the existing invariants or specifications are not strong enough, revises the specs and invariants, and tries again. This takes a long time because the issues are discovered one at a time, and each iteration through the cycle takes a long time.

One example of this showed up when we were trying to prove the user-level application property for the `echo hello world` application. When we started doing the proof, our syscall specifications had many limitations, where their statements were not quite strong enough to allow our proof to conclude that `echo hello world` typed at the UART console input will indeed produce the correct output. We made many iterations, together with the agent, in revising the specs, with each iteration taking on the order of an hour.

Multi-agent design review, described in §10.3, is helpful from this perspective, because it cuts down this slow iterative cycle, by finding potential specification issues faster (without having to go through a complete hour-long iteration), and finding many potential issues in one go (allowing them to be addressed all in a single iteration, instead of one at a time).

10.5 Agent memory is sticky and implicit

We have found that, while agents run, they make implicit design decisions and assumptions about what the goal is, what the proof should be like, what constraints exist on the specs and proofs they are working on, what the implementation intends to do, etc. It is infeasible to fully review the agent's internal assumptions, especially when using a coordinator agent to orchestrate many sub-agents. One failure mode that results from this is that, once an agent (or an agent with its sub-agents) makes some bad implicit assumption, it is hard to get the agents to complete the task at hand, and it is often unclear why the agents are unable to finish it.

To address this problem, we find it is useful to start agents from scratch. When an agent reaches a reasonable checkpoint—either finishing its work, or reaching some consistent unfinished state where it seems to be unable to make progress—we ask it to checkpoint its state into a project status document, and start the next agent fresh. The next agent reads the project status, along with general design documents and notes in the git repository, and tries to solve the problem with a fresh context. We review the project status to understand what implicit assumptions the agent might be making, and we also ask the fresh agent to explain to us what the status of the project is, what the issues are, and what its plan of attack is. This often helps uncover implicit assumptions that are preventing the agents from finishing certain proofs.

One place where we have found it hard to "clear" the agent's thinking is in the code that is already committed in the git repository. In particular, we found it is much easier to get agents to write new specs and proofs than to refactor or revise existing ones, and when agents are working with existing specs and proofs, it requires effort to get them to adopt a new design. Writing new specs and proofs tends to work well without much supervision; minimal prompting suffices. On the other hand, performing some refactor, even if relatively simple, requires careful prompting and supervision to ensure the agent does the right refactor, does not "reinvent the wheel" along the way, does not fall back to the original design, etc.

Finally, we have sometimes discovered that agents already "knew" of a substantial issue but failed to tell us about it until we explicitly asked. For example, towards the end of finishing up the entire kernel proof, we asked the agent whether all of the in-flight projects had been completed (expecting the answer to be "yes"), and to move those completed project documents to an archived directory. However, the agent came back telling us that, no, one of the projects was clearly documented as not being completed—file system crash safety. Indeed, it turned out that the project document for the file system crash safety effort said, as one of its many bullet points, that the crash safety theorem was vacuous (the theorem assumed that every time the system rebooted, it started with a fresh copy of the `fs.img` image from `mkfs`, rather than the state of the disk at the moment of the crash). This was written in the design document, but the design document had a long list of irrelevant low-level details, so we did not read it in full before the agent surfaced this issue when forced to decide whether the project was done.

10.6 File organization discipline

We find that it is highly effective to have a strict organizational structure for the files in our project's git repository. In particular, for every kernel function `F`, we have:

- a `SpecF.v` file that states the function's specification.
- a `CodeF.v` file that captures low-level facts about the RISC-V instructions that comprise the function itself (this is largely auto-generated by a Python script, written with the help of an agent, but the resulting `CodeF.v` file is still checked by Rocq just like every other file).
- a `ProofF.v` file that proves that the implementation of the function, as captured by `CodeF.v`, meets the precise spec as stated in `SpecF.v`.
- a `LinkF.v` file that links the proof of `F`, from `ProofF.v`, with proofs of any other functions that `F` calls; this ensures that all of the functions agree on the specs for all other functions.

The reason why we have link files is that our specification of a function `F` can be dependent on other specifications—namely, the specs of all of the other functions that `F` invokes. Formally, this is stated as a premise of the theorem about `F`'s execution—to conclude that `F` executes correctly, the caller of the theorem has to first prove the correctness of all the functions that `F` might call, with the expected specifications for those functions. A link file instantiates the theorem about `F`'s correctness by filling in these premises, using proven theorems about those called functions. The linking ultimately converges to a proof for the system initialization code that lives at address `0x80000000`, where the CPU starts executing at power-on.

In addition to having this family of files for every function, we also have well-defined files for each level of abstraction in the kernel. For example, we have a file defining the abstractions for disk blocks, for inodes, etc., and also files defining the invariants that hold on top of those abstractions.

The most substantial benefit of this structure is that it makes it clear to agents which files they are expected to modify, and which files they are expected to keep unchanged. While Rocq is ultimately responsible for checking that all of the proofs are correct, we find that this code structure helps guide agents in the correct direction. As a counterexample, early on in this project, we had a single file that would define the spec for a function, the abstractions it provides, the proof of its correctness, its invariants, and its dependencies on other functions. Agents would often get confused as to the rules we were expecting them to follow, and would change the spec of a function to make the proof easier, re-define the abstractions to make postconditions less useful (or even vacuous), etc.

A second benefit of this file structure is that it enables substantial parallelism in terms of build times. By having each function proof be independent of any other function proof, `make` can run Rocq in parallel on all function proofs at the same time; these proofs comprise the bulk of the CPU time required to do a build.

A third benefit of this approach is that it allows for cleaner separation of changes between concurrent agents. When agents are working on different files, they can first reach agreement on what the specs are going to be at the boundary between them, and then independently work on the proof files for their functions without conflicting with any changes from other agents.

10.7 Performance optimization

Fast proof checking enables agents to iterate more quickly when developing proofs. To ensure that our proofs have reasonably fast proof-checking times, we continuously prompt an agent to improve performance, profile the proofs to find expensive statements, optimize, etc. We find this to be effective: this approach finds many performance pitfalls, and fixes them.

One class of optimizations made possible by the fact that we use agents is writing highly specialized proofs that do not rely on proof automation. This trades away proof automation that would be important for human proof engineers (e.g., a single `iFrame` tactic that does a lot of search to do a big proof step, or a single `set_solver` tactic that solves a large family of set-theoretic goals) in favor of special-case proofs at each point (spelling out exactly how to prove resource ownership instead of `iFrame`, or spelling out exactly the proof argument for a set-theoretic goal without using `set_solver`), which improves performance of proof builds.

Occasionally, when we asked the agent to do performance optimization, it uncovered new avenues for optimizing proof performance—in other words, the agent's behavior was non-deterministic in terms of what the agent would find each time, and it was worthwhile to keep asking it to do the same task multiple times. For example, at one point

it decided to profile the OCaml implementation of Rocq itself, found some sources of quadratic performance blow-up, and figured out a way to avoid them by being careful about what tactics it used in its proofs.

Some Rocq proof files took several minutes to compile, and iteratively fixing up those proof files through an edit-recompile cycle took a long time, especially when the broken part of the proof was towards the end of the file. The agents were not familiar with the interactive style of using Rocq through `rocq repl`, so instead we developed a tool that cached the state of `rocq repl` for a large proof file, and incrementally re-checked a modified file by rewinding the `rocq repl` process to the common prefix of the old and new files, and replaying new proof commands from that point [107].

10.8 Orchestration for large projects

When undertaking a large change that requires significant design effort, we use highly capable models (such as Fable in the case of Claude Code) to drive the top-level agent's loop, and we ask the agent to start sub-agents to do specific well-scoped tasks (e.g., using Opus or Sonnet). This is analogous to how agent-based software development workflows use agent orchestration, and works well for proofs too. We find that highly capable top-level agents do a good job of ensuring that the specs are meaningful, and that the overall project will converge, whereas agents running less capable models like Opus and Sonnet are adequate for executing individual proof tasks when they do not have to design the overall proof plan.

On the other hand, for smaller-scale tasks that do not require high-level design decisions, we use Opus or even Sonnet agents directly. For example, most of the performance profiling, as described above, was done by Opus and Sonnet agents, especially once the optimization techniques had been discovered and described in the shared notes in the repo, so that the agents simply needed to match the performance problem observed in some proof against the well-known optimization techniques.

10.9 Kernel changes

One challenge of verifying the xv6 kernel at the level of its ELF binary is that any change to the xv6 source code changes the entire ELF binary. The compiler might choose different instructions, different relative offsets, the symbol addresses might move around, etc. This is one aspect of the proof where agents make the approach viable: it is unlikely that human proof engineers would be interested in fixing up all of the proofs after every recompile. On the other hand, we have found that, each time we made a change to the xv6 source code and recompiled it, an agent was able to fix up all of the proofs within several hours. Throughout these iterations, we asked the agent to develop tools to make future iterations cheap, such as: making as many address references as possible symbolic, referring to symbols from the ELF symbol table; developing scripts to automate generating the `CodeF.v` files from the ELF dump as opposed to writing them by hand; developing scripts to automate replacing PC-relative offsets in instructions throughout the proofs, instead of having an agent reason about each broken proof step iteratively; etc.

11 Bugs found

In the process of specifying and proving the xv6 kernel, we found 13 bugs, summarized in Figure 29: one in the Sail RISC-V model that we use as our specification of the hardware, one in Sail's Lean extraction of that model, 10 in the xv6 kernel itself, and one in the `grep` user program. The rest of this section describes each of them in turn.

Sail model bug: A/D writeback. We discovered a bug in the Sail RISC-V model itself, when we could not fully prove the correctness of our page-table and TLB invariant. The bug comes down to the hardware model's implementation of the writeback of A/D (accessed/dirty) bits into the page-table entry. When the Sail model decided to write the A/D bits back into the actual page table, it failed to check whether the page-table entry was still valid, because it did a blind write rather than an atomic read-modify-write operation. This meant that one CPU could have cleared a page-table entry (e.g., in order to free a page of memory), and then another CPU would inadvertently "resuscitate" this mapping by doing a blind write to the PTE as part of the A/D writeback. We submitted a fix for this issue to the Sail RISC-V model developers, who agreed that this is a real issue. The fix involves doing an atomic read-modify-write to perform the A/D writeback.

Sail Lean extraction bug: Short-circuiting semantics. When we ported our proofs from Rocq to Lean, the agent discovered that the Lean extraction of the Sail RISC-V model incorrectly implements the semantics of boolean operators. In particular, Sail defines the semantics of and and or operators as short-circuiting: e.g., when the model says *A* and *B*, if *A* is false, then it does not run *B*. However, the Lean extraction of the Sail model implemented eager evaluation of both sides of the boolean operators: the Lean representation first evaluated both *A* and *B*, together with their side-effects, and then chose the resulting boolean value. As a consequence, if *B* had side-effects when *A* returned

Bug	Component	Consequence
A/D writeback was a blind write	Sail model	One CPU could resuscitate a page-table entry that another CPU cleared
Incorrect short-circuiting boolean semantics	Sail extraction	Wrong semantics of boolean operators in the Lean version of the CPU model
`iput` freed inode before dropping `struct inode` reference	fs	Racing `iput` and `ialloc` could orphan an on-disk inode until the next reboot
Creating files in disconnected directories	fs	`..` could name a free inode, and the kernel panicked because its type was not `T_DIR`
Overflowing `nlink`	fs	Too many links to one inode overflowed `nlink`; unlinking it panicked
Scheduler did not reset `push_off`'s `intena` flag	scheduler	Scheduler re-enabled interrupts, and could miss a wakeup between scanning the process table and `WFI`
Partial `writei` was not logged	fs	Partially written block was never logged, so reads changed once the buffer cache recycled it
Missing icache fence when running user code	exec, trampoline	Stale instruction cache entries from an earlier process could be executed after `exec`
`freeproc` updated `p->parent` without `wait_lock`	proc	`p->parent` was written without the lock that protects it
Inconsistent UART TX FIFO handling	uart	`printk` racing with console writes could overflow the TX FIFO
Boot code did not enable `ADUE`	boot	On hardware that follows the spec, the kernel did not handle the A/D traps it then received
`kill(0)` marked an unused `struct proc` as killed	proc	A future process was killed immediately after `fork`
`grep` stopped at a line longer than its buffer	user (`grep`)	A line of 1023 or more bytes silently ended the search

Figure 29: Bugs found during verification.

false, *A* and *B* would be executed incorrectly in the Lean version of the Sail model. We fixed this by changing the Sail tool to generate correct Lean code, so that the expected semantics of boolean operators are faithfully represented.

Kernel bug: `iput` could lose an inode. The kernel had a bug in reclaiming inodes once the inode's refcount reached zero; Figure 30 shows the code. When dropping the last refcount (from 1 to 0) on an unlinked inode, the `iput` function first marked the on-disk inode as free while holding the lock on the `struct inode` (`ip->lock`), then released that lock, and only then acquired another lock (`itable.lock`) to actually drop the refcount on the `struct inode`. The problem is that, between releasing `ip->lock` and acquiring `itable.lock`, the inode number was already available for allocation, while the `struct inode`, representing an inode cache entry, still carried `iput`'s reference. Thus, a concurrent file creation could allocate the same inode number and obtain a second reference to the same `struct inode`.

This bug manifested itself as a race. Consider a process P1 unlinking a file: its `iput` sets the inode's type to 0 (marking the inode as free), but P1's `iput` still holds its reference to `ip`, and has yet to decrement `ip->ref`. Before the decrement, there is a short period of time during which P1 holds no locks (between releasing `ip->lock` and acquiring `itable.lock`). Thus, some other process P2 creating a new file may allocate the inode (since its type is 0), setting the type to some non-zero value (e.g., `T_FILE` for a regular file). P2's `iget` in `ialloc` finds the same `struct inode` in the inode cache, since P1's `iput` has not yet decremented `ip->ref`, and `ip->ref` becomes 2. Now P2 may `unlink` the file; its `iput` decrements `ip->ref` from 2 to 1, but it will not free the on-disk inode because it is not dropping the last reference. Then, P1's `iput` finally decrements `ip->ref` to 0. At this point, the on-disk inode has `type` $\neq 0$ and `nlink` $= 0$. The inode is orphaned without a crash. A reboot will fix it through `ireclaim`, but before then the inode is lost.

The fix, shown on the right of Figure 30, was to free the inode by its inode number, after decrementing `ip->ref`, rather than by its `struct inode` pointer. We were initially not confident in the fix, because it required the kernel to operate on inode numbers directly, which is not the case for any other file system code, and the fix had the potential to affect some other part of file system correctness that we might not have thought about. However, after we proved the kernel correct with the fix, we had sufficient confidence to commit this fix in the upstream xv6 repository.

```
void
iput(struct inode *ip)
{
  acquire(&itable.lock);
  if (ip->ref == 1 && ip->valid &&
      ip->nlink == 0) {
    // ref == 1: nobody else can hold ip->lock
    acquiresleep(&ip->lock);
    release(&itable.lock);

    itrunc(ip);
    ip->type = 0;  // mark on-disk inode free
    iupdate(ip);
    ip->valid = 0;
    releasesleep(&ip->lock);
    // P1 holds no locks here.  P2's ialloc() can
    // reuse the inode, and if P2 then unlinks the
    // file, it will not free the inode, because
    // there is an outstanding refcount from P1.
    acquire(&itable.lock);
  }
  ip->ref--;
  release(&itable.lock);
}
```

```
void
iput(struct inode *ip)
{
  acquire(&itable.lock);
  int last = (ip->ref == 1 && ip->valid &&
              ip->nlink == 0);
  uint dev = ip->dev, inum = ip->inum;
  if (last) {
    // ref == 1: nobody else can hold ip->lock
    acquiresleep(&ip->lock);
    release(&itable.lock);

    itrunc(ip);    // on-disk type stays non-zero
    ip->valid = 0;
    releasesleep(&ip->lock);

    acquire(&itable.lock);
  }
  ip->ref--;
  release(&itable.lock);
  if (last)
    ifree(dev, inum);  // set on-disk type to 0
}
```

Figure 30: Simplified code of `iput` in xv6 with the bug (left), showing where a concurrent file create and unlink by another process P2 can interleave, and after the fix (right). In the fixed code, `ifree` marks the on-disk inode `inum` as free by setting its type to 0.

Kernel bug: Creating files in disconnected directories. If a process's cwd was a directory that had been deleted, and the parent directory was also deleted, then `..` referred to a free inode, and trying to access `..` caused the kernel to panic because the inode type was no longer `T_DIR`. The fix was to make `namei` return an error if it is traversing a directory whose own link count is zero. Similarly, we fixed `create` and `link` to check that they are not creating a new filename in an unlinked directory.

Kernel bug: Overflowing `nlink`. xv6 did not check for overflow on the inode's link count. As a result, creating many links to a single inode could overflow that inode's `nlink` and wrap it around to a negative value. The implementation of `unlink` panics if it finds `nlink` < 1. The fix was to check for overflow, and return an error if creating a new link would cause the inode's `nlink` to overflow.

Kernel bug: Scheduler did not reset `push_off`'s interrupts-enabled flag. When a kernel thread context-switches to the scheduler using `swtch`, the scheduler inherits the per-CPU flag tracking whether the top-level `push_off` had interrupts enabled or disabled (this flag determines whether the last `pop_off` will re-enable interrupts on the CPU). The scheduler needs to have interrupts remain disabled: it checks the entire process table to see if there are any more runnable processes, and if it makes a complete loop finding no runnable processes, it goes to sleep using the `WFI` instruction. If an interrupt were to come in between the scan and the `WFI`, and the interrupt woke up some process, the scheduler would not notice. However, the scheduler forgot to reset this per-CPU flag when it resumed execution after context-switching from a kernel process thread, which inadvertently caused it to re-enable interrupts when the scheduler released the spinlock protecting the currently running `struct proc`. The fix involved explicitly clearing this bit in the scheduler before releasing any locks.

Kernel bug: Failing to log a partial file write. In xv6, `sys_write` calls `writei`, which needs to copy data from the user-supplied pointer into the file's data block. To do so, `writei` gets the data block, and then calls `copyin` to safely copy data from a user-space pointer into the data block. The user buffer could span a page boundary; in this case, `copyin` validates the mapping of the first page, copies the bytes from there, and then proceeds to validate and copy data from the second page. If the second page turns out not to be mapped, then `copyin` returns an error after having already copied data from the first page. On receiving an error from `copyin`, `writei` returned an error back to `sys_write`, without putting the partially updated data block into the write-ahead log. This caused the file's data block to enter an unstable state: as long as it remained in the buffer cache, reads would return the partially written data, but once the buffer was recycled, reads would start returning the old data. We found this bug while proving that

every disk buffer write is logged to the write-ahead log. The fix was to have `writei` log the data block even when `copyin` returns an error.

Kernel bug: Missing icache fence when switching to a user process. RISC-V does not require the instruction cache to be coherent with writes to memory. This is not a concern for the kernel's code, because the kernel code is loaded at boot time and is never modified afterwards. However, user code *does* get loaded dynamically during `exec`, and gets stored in dynamically allocated pages of memory. As a result, after the kernel runs `exec` and starts executing the process in user space, the instruction cache could still contain old cached entries back from when this same physical page of memory contained the code of a different user process. To fix this bug, we added `fence.i` to the xv6 kernel's `userret` trampoline, before it returns to user space.

Kernel bug: `freeproc` failed to hold `wait_lock` while updating `p->parent`. xv6 uses a separate lock, `wait_lock`, to protect the parent field of a `struct proc`, `p->parent`, instead of using the spinlock associated with the rest of the process, `p->lock`. The reason is that operations that deal with `p->parent` need to operate on multiple processes at a time, such as re-parenting an orphan process to `init` (PID 1). However, the `freeproc` function modified `p->parent` without holding `wait_lock`. We found this while proving `freeproc`; the proof did not have the ownership of `p->parent` needed to write to that field, despite holding the `p->lock` spinlock, since ownership of the `p->parent` fields lived in the lock invariant of `wait_lock`.

Kernel bug: Inconsistent UART TX FIFO handling. The xv6 kernel was not consistent about its locking discipline when writing data to the UART. There were two ways that the kernel would write to the UART: one coming from user processes writing to a console file descriptor, and the other coming from the kernel's calls to `printk`. Both paths checked whether the UART's TX FIFO was full by checking a status register in the UART, and if it was not full, wrote bytes to the UART's TX FIFO register. However, the two were not synchronized with one another, which meant that it was possible for the TX FIFO to overflow when the two raced. This showed up in the proof when we could not assign consistent ownership of the UART's TX FIFO state to either of the two lock invariants. Each was independently proven to be correct, but when we tried to compose `printk` together with user-space processes writing to the console, the adequacy theorem could not assign the same device ownership to both invariants. We fixed this by using a single spinlock to protect the UART TX FIFO state (i.e., both `printk` and user console writes grab this spinlock, check if the TX FIFO is available, and if so, write data to it).

One reason the xv6 kernel had this bug is that xv6 is intended to be a teaching operating system, where students modify its source code through a series of lab assignments. Students inevitably make mistakes, including bugs that lead to deadlock, corruption, etc. Thus, it was important for xv6 to ensure `printk` can run and print out debugging information to help the student, even if, say, the console lock is already held by some deadlocked process. On the other hand, for the verified version of xv6, the `acquire` specification enforces the lock-ordering discipline and the proof rules out corruption, so this aspect of `printk`'s design was no longer important.

Kernel bug: Boot code did not enable `ADUE`. RISC-V provides two ways to manage page-table accessed and dirty bits: either hardware-managed ("Svadu"), where the hardware writes back A and D bits into page-table entries when pages are accessed or modified through the TLB, or software-managed ("Svade"), where the hardware traps into the kernel in order to update A and D bits. The RISC-V specification, and the Sail model, state that by default the system starts in software-managed mode, and switching to hardware-managed mode requires setting the `ADUE` bit in the `menvcfg` register. However, QEMU starts in hardware-managed mode by default, for backward-compatibility reasons. xv6 was tested only on QEMU, and failed to set `ADUE` in `menvcfg`. We discovered this when the proof required reasoning about traps on accesses to every virtual memory address; the xv6 kernel did not handle these traps. The fix was to set the `ADUE` bit in `menvcfg` in xv6's machine-mode boot code.

Kernel bug: `kill(0)`. As part of specifying the system calls, we discovered that the implementation of `kill(0)` set the `killed` flag on the first unused `struct proc`, which happened to have a `p->pid` of zero. This did not violate any internal kernel safety invariants, but it did make for a surprising specification of `kill`, because, with this buggy implementation, some unspecified future process would get killed immediately after `fork`. We fixed the implementation of `kill` to do nothing when invoked with a zero PID.

User-program bug: `grep` silently dropped input. xv6's `grep` reads its input into a fixed 1024-byte buffer, repeatedly calling `read` to fill up the buffer until it sees a newline. Each repeated call to `read` asks the kernel to read the number of bytes corresponding to how much space remains in the buffer. When the buffer reaches 1023 bytes of data with no

newline, the loop calls `read` with a zero length, which causes the kernel to return 0. The code interprets that as an end-of-file indicator, and `grep` exits.

To fix this bug, we changed `grep` to continue reading bytes until the next newline. Our fixed version of `grep` does not match on lines that are at least 1023 bytes long (since we did not want to introduce unbounded memory allocation), but it does continue to match subsequent lines.

12 Implementation

The implementation effort took 102 days and 7,715 commits. Figure 31 breaks the development down by component. Of the 1,597,440 lines of Rocq in the tree, 331,166 lines are generated by tools and never edited: the Sail RISC-V semantics compiled to Rocq, the linked kernel and user ELFs dumped to Rocq data, the instruction-decode layer re-derived from that dump, and the reference-implementation results that the conformance tests are checked against. The remaining 1,266,274 lines are written by agents.

265,349 lines are not about xv6 in particular. 42,948 of those are the logic itself: the Iris language instance over the Sail execution monad, the points-to assertions for registers, memory, and devices, the per-node proof interface, the TSO memory model, and adequacy. 104,989 lines are instruction rules: the per-instruction WP leaves and the step engines they compose, for machine mode, supervisor mode under the kernel's translation regime, and the trampoline page. 17,411 lines are RISC-V architecture theory with no separation logic in it (Sv39 page-table trees and walks, the `mstatus` transforms, decode totality, bitvector arithmetic), and 4,535 lines are the device models: the UART, the PLIC, and the virtio disk, as part of the machine model beside the Sail core. Two more rows are about user-mode execution in general: 41,018 lines prove that arbitrary user code runs safely under a user page table (§6.13), and 54,448 lines are the verified-user tier that runs a program with a known image instruction by instruction, together with the user-side heap and descriptor resources (§7).

724,667 lines are about the xv6 kernel. 145,938 lines define resources for kernel state: the process table and scheduler protocol, the buffer cache, the inode cache and its escrow, the log, pipes, spinlocks and sleeplocks, the kernel and user page tables, the file system's state, durability, and crash predicates, and the pure on-disk semantics those read. 18,718 lines are the driver-side proof plan for the devices: the DMA lease and the keyed queue protocol the disk driver shares with the device thread, the kernel's plan for the PLIC, and the three device threads with the invariants they run under.

The function specifications are 85,030 lines (`Spec*.v` plus the statements they import: the abstract file system state, the per-syscall definitions and log budgets, and the user/kernel trap contract). 445,970 lines are whole-function proofs, 3,288 lines link each function proof to the proofs of the functions it calls, and 25,723 lines boot the system—the M-mode carve of the boot image, the ghost allocation of every subsystem, the era-0 facts read off the `mkfs` image—and assemble the top-level theorem.

263,801 lines are the user side: 135,960 lines of proof about `init`, `sh`, `cat`, `echo`, `sync`, `grep`, `seccomp`, and the `ulib` routines they call, and 127,841 lines of application claims—the pure session models of what the user may type and what the console must show, the Iris claims over the abstract file system and console, and the application theorems. The rest of the written Rocq is 5,728 lines of tests, pilots, and superseded drafts kept as records; 6,583 lines of conformance-test proofs (§9); and 146 lines of platform hooks for the Sail model.

Outside Rocq, the development carries 10,358 lines of Python (the ELF dumper, the decode-layer and code-catalog generators, the conformance-test harness, and the reporting tools), 12,065 lines of agent-written RISC-V assembly for the conformance-test images, and 165,505 lines of Markdown design notes in 238 files.

Trusted definitions. Rocq's kernel checks the proofs, so most of the lines shown in Figure 31 are not trusted, as long as the top-level theorem statement is correct. To measure how many lines are required to define the meaning of the top-level theorems, we used Rocq's `Print All Dependencies` command to track down what definitions are transitively used in the theorem statements (as opposed to their proofs). Figure 32 shows the results.

The generic adequacy theorem, `xv6_power`, which proves that any pure property that can be derived from the invariants maintained by xv6 holds on every trace-level execution according to the operational semantics, depends on 795 definitions spanning 6,660 lines in our proofs, plus external dependencies (the Sail model, the dump of the ELF kernel, the stdpp and Iris libraries, etc.). The statement of this theorem includes some of the CSL machinery, because it talks about any pure invariant that can be derived from CSL invariants, which inherently needs to talk about Iris CSL properties.

We also have a specialized instantiation of the above generic theorem, `xv6_fs`, where we prove it for a specific trace property: that the file system remains consistent (across all concurrent executions, all crashes, etc.). This adequacy

Component	Files	Lines
Written: not specific to xv6		
Program logic: Iris over Sail, memory model, adequacy	53	42,948
Instruction rules: WP leaves and step engines	109	104,989
RISC-V architecture theory	23	17,411
Device models (UART, PLIC, virtio disk)	3	4,535
Arbitrary user-mode execution	33	41,018
Verified user-execution tier	59	54,448
Written: the xv6 kernel		
Kernel abstractions and invariants	166	145,938
Device driver protocols and device threads	8	18,718
Specifications (`Spec*.v` and their vocabulary)	239	85,030
Whole-function proofs (`Proof*.v`, ...)	322	445,970
Linking (`Link*.v`)	209	3,288
Boot, system assembly, top-level theorem	37	25,723
Written: user programs		
User-program proofs (`Uk*.v`)	97	135,960
Application claims and theorems	166	127,841
Written: other		
Tests, pilots, superseded drafts	11	5,728
Conformance tests	13	6,583
Sail platform hooks	2	146
Generated		
Sail RISC-V model in Rocq	2	59,887
Kernel image in Rocq	5	60,654
User program images in Rocq	28	46,502
Instruction-decode layer	224	113,557
Captured QEMU, JH7110, and CVA6 results	442	50,566
Total Rocq		**1,597,440**

Figure 31: Size of the development, in lines of Rocq. "Generated" files are produced by tools and are never edited by hand.

What the statement rests on	`xv6_power`	`xv6_fs`	`union`
Sail RISC-V model, compiled to Rocq	21,585	21,212	21,212
Machine and shared-memory model	1,535	1,207	1,202
Device models (UART, PLIC, virtio disk)	1,806	1,797	1,797
Initial disk image (plus generated Rocq representation of a 2.0 MB binary file)	37	37	37
Kernel image (generated Rocq representation of a 41 KB binary file)	—	—	—
Iris ghost state and CSL specifications	1,788	—	—
The conclusion statement (FS consistency, UART console trace, etc.)	1,494	994	1,591

Figure 32: Lines of definition each adequacy statement unfolds to, by category.

corollary unfolds to 499 definitions spanning 4,035 lines. These definitions are notably pure—they do not mention any CSL machinery at all—because the proof of this corollary establishes that the xv6 CSL invariants imply the pure file system consistency property, and therefore the theorem statement itself does not need to refer to CSL in any way. On the other hand, this corollary does include additional dependencies that define what file system consistency means.

Finally, the application-level theorem of §2, `union`, instantiates the same generic theorem with a different trace property: the console output that a shell session may produce when the user types any sequence of commands of the forms `echo ...`, `echo ... > f`, `cat f`, pipelines such as `echo ... | cat | grep w`, using `sync` to flush file system changes to disk, and using `seccomp` to run unverified binaries in a sandbox, across any number of power cycles. This theorem unfolds to 630 definitions spanning 4,627 lines. Its statement is free of CSL machinery, since the conclusion is a property of the bytes on the console UART alone; the additional definitions describe what each command, and each stage of a pipeline, may print.

Invariants. The kernel's shared state is owned by Iris invariants, allocated in 30 distinct namespaces. Thirteen of them belong to the file system: the log's block bytes, an escrow through which a `sync` that finds a commit in flight collects its result from that commit, the block bitmap, the inode table, the inode reference escrow, the per-inode

escrow, the free-inode pool, the inode region, the file-table top, the superblock, and three families of per-object *transit boxes* (invariants for reasoning about shared memory locations that are accessed under different locks, which is tricky to reason about under weak shared memory). Four belong to the device fabric (§3): the UART, the PLIC, the virtio disk, and the external-interrupt pins the PLIC drives into each HART. One is the kernel page table; two belong to the process layer (the per-lock invariant, instantiated once per spinlock, and the process-slot availability ghost); and four are machine-level resources the boot path configures and every later proof depends on: the PMP configuration, the timer comparator, the boot handshake, and the power/crash state. One further invariant, which holds the trace property, belongs to the whole-system statement rather than to any one part of the kernel. The remaining five support application-level proofs (§7): the application invariant (§8), the user-level half of each file descriptor's offset (Figure 23), an escrow for the right to read console input, an invariant for concurrent user reads/writes of a pipe, and one for a pipe shared with a `seccomp`-sandboxed program, whose reads and writes follow no protocol.

Build time. Building the whole development takes about 81 minutes on a 24-core virtual machine with Intel Xeon E5-2699 2.2 GHz vCPUs and 96 GB of memory: 2 minutes for the Sail model and the ELF dumps, 56 minutes for the Iris proofs, 17 minutes to run `Print Assumptions` on the top-level theorem to confirm the set of assumed axioms, and 6 minutes to check the conformance-test proofs.

Porting to Lean. After we had finished doing all of this verification work in Rocq, we asked an agent to port the whole verification framework and proofs to Lean. This required some supervision, but far less than the original development. The resulting Lean port was about half the size of the Rocq development, in terms of lines of proof, and it compiled (checking all of the proofs) about 5× faster, through a combination of cleanup, simplifications, and better optimizations in Lean itself. (This paper, including all of the reported numbers, describes the original Rocq development.)

One surprising finding is that, when one of the authors of this paper switched the git repo from Rocq to Lean, the other author barely noticed (aside from the performance improvements), since the level of discussion with the agent was far above the specific syntax differences between Rocq and Lean; the agent was able to switch all of its work from Rocq to Lean transparently.

12.1 Changes to xv6

We made almost no changes to xv6 itself, other than fixing the bugs described in §11. Even in tricky cases, we forced the proof to work with the code, rather than changing the code. One example is the file reference counting, which does not overflow for a subtle reason. Another example is `kexec` not causing a panic when remapping the trampoline page at the top of the address space.

We made some changes where the xv6 kernel was purposely sloppy to support students doing lab assignments on top of xv6. In particular, it was important for xv6, as used for class assignments, to ensure that `printk` continues working even if the students modify the kernel, run into a bug, and call `printk` when the rest of the kernel is corrupted. To this end, `printk` tried to avoid acquiring locks, at the risk of corrupting the UART TX FIFO (see §11). In our verified xv6 kernel, the proof forces every caller of `printk` to follow the lock-ordering discipline, so we changed the implementation to be correct by holding locks.

Along the same lines, we removed support for `procdump`, which printed a listing of currently running processes when the user entered `Control-P` on the UART console. This, too, existed to support student debugging, and to implement a cheap version of the `ps` command. However, `procdump` did not acquire appropriate locks, again in order to ensure that some version of `procdump` can be executed even if the student's lab assignment has deadlocked the kernel. We removed the support for the `Control-P` special input handler in the UART console.

We proved that many of the calls to `panic` were unreachable because the kernel maintains all of its invariants, and therefore many assertions about kernel data structure consistency never fail. However, the kernel still has some calls to `panic` that represent running out of resources. For example, the kernel panics if it runs out of in-memory `struct inode` entries. On the other hand, the kernel correctly handles running out of memory without panicking.

We also deleted some lines of code where the proof helped us realize they were superfluous. One example is that `kexec` did an explicit check on the `argc` value to ensure that it did not exceed the maximum allowed number of arguments. It turns out that `sys_exec`, which is the caller of `kexec`, already performed this check, and moreover, the check inside `kexec` was not sufficient (it was off by one in an unsafe direction).

Finally, we fixed PID wraparound in xv6; the implementation previously assumed that a 32-bit integer PID would never overflow. To prove that PIDs uniquely identify processes, and that PIDs are never negative or zero, we modified

the implementation to check, at allocation time, whether the PID has wrapped around or is already in use by an existing process.

12.2 TCB

The theorem about the xv6 kernel's correctness requires trusting several components, namely:

- The Sail model of RISC-V.
- Sail's Rocq backend that generates a Rocq definition based on the semantics of RISC-V defined in the Sail language.
- The Rocq proof-checking kernel.
- The ELF dumper, which is a Python script that translates the kernel's ELF binary into a Rocq file representing all of the bytes in the ELF image, along with the text and data sections of the kernel (since the semantics assumes the kernel is already loaded in memory based on the ELF program headers, rather than loading the literal ELF binary into DRAM).
- The `Print Assumptions` command that confirms the axioms assumed by the top-level adequacy theorem (which, in our case, are the standard logical functional-extensionality axiom, as well as two axioms required by the Sail model to represent its behavior for memory reservations, used by the LR and SC instructions).
- The model of how the system executes, which is built on Sail's semantics, and includes a model of weak shared memory (TSO with load-load reordering) with handling of exclusive read-modify-write operations, introduced by us, as well as models of devices (UART, PLIC, and virtio disk) that we defined, and a model of power handling (power-off and power-on) to represent crashes. We validate this model using conformance tests (§9), but it is not a proof that considers all possible cases.

There are also a number of notable components that are not in the TCB:

- The kernel implementation, including both the C code and all of the assembly: the boot code, context switching, hardware configuration, page-table setup and management, switching to user space, taking interrupts from user space, taking interrupts while running the kernel, TLB A/D writeback, I/O memory fences when dealing with disk DMA regions, etc.
- The compiler, linker, assembler, and all other parts of the kernel build process that lead to producing a single kernel ELF binary.
- The correctness of the initial file system image, as produced by `mkfs`.

One shortcoming of the Sail model is that it is over-specified in some cases; for instance, it models the TLB as a 64-entry direct-mapped cache, rather than allowing any associativity or different sizes. We also found (and fixed) the PTE A/D writeback bug in the Sail model.

The Sail RISC-V model comes with an extensive configuration file format that requires any user of the Sail model to define the specific features they want to include. We chose a specific set of features that correspond to the platform targeted by the xv6 kernel (e.g., not including B, V, FP, and control-flow integrity features, among others).

13 Evaluation

This section answers the following questions:

- How much agent work was required to verify xv6?
- How much effort is needed to adapt the proofs when the xv6 source code changes and the ELF kernel binary is recompiled?
- How much effort is required to adapt to changes in the Sail RISC-V model?
- What issues were uncovered by model conformance tests?

Elapsed time	102 days
Git commits	7,715
Agent sessions / working directories	492 / 13
Sub-agent runs	2,765
Human prompts	6,058
total words / median words per prompt	160,091 / 13
Agent messages / tool calls	489,448 / 501,905
Claude running time (summed over sessions)	2,153 h
wall-clock (union of sessions)	1,272 h
Output tokens	235.3 M
Input tokens (171.8 B cached)	174.3 B

Figure 33: Development effort, extracted from the Claude Code transcripts.

13.1 Agent effort

To report on the agent effort required to verify xv6, we extracted data from both authors' Claude Code transcripts, which record every message, its timestamp, and its token usage; Figure 33 summarizes them. Unfortunately, part of the record is lost, because Claude Code deletes transcripts older than 30 days by default. Roughly 9% of the sessions are missing, most of them from one window of about two weeks early in the development process. The rest of the record is nearly complete, including the project's first two weeks.

Work was spread over 492 agent sessions in 13 working directories—separate checkouts of the same repository, so that several agents could work at once (§10)—and those sessions spawned a further 2,765 sub-agent runs. Across all of them we typed 6,058 prompts, totaling 160,091 words. The typical prompt is short: the median is 13 words and the mean 26.4. Most of them are of the form "finish the `consoleintr` proof" or "git pull the latest changes; specify and prove `sys_close`"; the long ones are design rulings, where we tell the agent what abstraction to use, or reject one it proposed. A further 2,385 prompts were generated by the harness rather than by us, continuing a session that had been told to keep going with `/goal`.

Against those 6,058 prompts, the agents produced 489,448 messages and made 501,905 tool calls, and Claude was actively running for 2,153 hours. Because up to 8 sessions ran concurrently, that is 1,272 hours of wall-clock time. ("Actively running" excludes gaps longer than five minutes between consecutive messages in a session, so a session left open overnight is not counted as active.) The token cost was 235.3 million output tokens, of which 57.6 million were thinking tokens, against 174.3 billion input tokens—almost all of them (171.8 billion) served out of the prompt cache. Claude's running time includes the time it spent waiting for Rocq, which is substantial.

This development cost less than $3,000 in Claude subscriptions (for multiple Max 20x plans), plus several multi-core development VMs on AWS and GCP.

13.2 Re-proving after kernel changes

The proofs are about a specific kernel binary, so any change to the xv6 sources moves symbol addresses and re-encodes call targets, and every proof that names an address is affected. The development was updated ("bumped") to a new version of the xv6 source code 26 times.

Almost all of the changes are mechanical. Across the 26 commits that updated the development to a new version of xv6, 1,544,180 lines of generated code changed—the kernel dump, the instruction-decode layer, the user-program dumps—along with 118,420 lines of written specification and proof, a median of 2,822 lines per bump.

To measure what a bump costs, we located, for each bump, the session in which the agent made that commit and measured from the prompt that asked for the bump to the commit. We could do this for 25 of the 26 commits; the transcript of the remaining one was pruned by Claude Code. The median bump took 35 minutes of agent time and 2 prompts; the cheapest took 2 minutes; the most expensive, 12.2 hours, was one where the C source gained a check (so the specification moved, not just the addresses) on top of a relayout that shifted most of the kernel's symbols. In total the 25 measured bumps account for 39.5 hours, 157 prompts, and 1.8% of the project's agent time.

13.3 Re-proving after Sail model changes

During the development of this project, we updated the Sail RISC-V model 3 times: adding a fix for the PTE A/D writeback bug, changing how Sail extracts axioms, and tagging instruction vs. data loads to enable modeling a non-coherent icache. The PTE A/D writeback fix required significant effort to adopt, because it brought a number of other updates to the Sail model along with it, and because it required revising the page-table proofs. That bump required 10.9 hours of agent time. Updating the axiom extraction required 1.7 hours, and the tagging change 0.8 hours.

13.4 Model discrepancies

We generated a total of 77 test cases covering behaviors of the CPU, page tables, interrupts, shared memory, CLINT, PLIC, UART, and the disk. The JH7110 and CVA6 do not implement the virtio disk device, and their UART devices are different from the one in QEMU and our model, so we did not run those tests on the JH7110 or CVA6. The JH7110 also implements an earlier revision of the RISC-V specification, which means that it does not have certain CSRs that xv6 uses (and that QEMU implements), and the JH7110 does not implement hardware writeback of PTE A/D bits.

Comparing the runs generated by these tests on our three reference implementations against the model surfaced 35 discrepancies; we describe the main ones below.

The UART, PLIC, and disk tests uncovered several issues with our model of those devices. To give a few examples: the UART device exposes several single-byte registers at consecutive addresses; the model failed to consider the possibility of a multi-byte read accessing the address of one of the UART's single-byte MMIO registers. The PLIC model failed to correctly implement priority thresholds. The disk model failed to correctly return the length of a used ring slot. xv6 did not depend on any of these features, but conformance tests found these differences between QEMU and our model, and after confirming what the correct behavior should be, we fixed these issues. The PLIC of the JH7110 matched our model in some cases, but differed in cases where the PLIC had more interrupt sources; our model was initially intended to match QEMU.

The disk test also uncovered a more serious issue with our disk model: our disk model executed all submitted disk requests in order, whereas the QEMU device can execute requests out of order (and indeed our tests observed such reordering). We changed our model to allow executing requests in any order, which required updating the invariant used by the xv6 disk driver, as well as the associated proofs. The file system already assumed that writes could be reordered, because this was the spec of the `virtio_disk_rw` function.

The CPU tests uncovered several cases where the initial register contents on QEMU differed from our model. These were all "benign" differences: as one example, QEMU reports in the `misa` register that it supports the hypervisor feature (H); our model does not support the hypervisor feature, and does not report this bit in `misa`. Similar issues showed up with the initial register contents on the JH7110.

The shared-memory tests uncovered the fact that QEMU can generate shared-memory accesses that are not sequentially consistent (we ran QEMU on an x86 machine, which implements a TSO shared-memory model). Our initial model of shared memory was sequentially consistent, and could not reproduce that memory behavior. We have since extended our shared-memory model to TSO with load-load reordering, which can represent this behavior.

The CPU tests observed that the instruction cache on the JH7110 is non-coherent: self-modifying code keeps executing old instructions until the machine executes `fence.i`. QEMU did not exhibit such behavior. We proved that our model of a non-coherent instruction fetch captures both of these behaviors.

The tests also uncovered a difference between the QEMU TLB behavior and the behavior of the Sail model. Sail's RISC-V model implements a 64-entry direct-mapped (non-associative) TLB. One of our page-table tests observed a difference between QEMU and the model because the model forced eviction of a certain TLB entry and re-walked the page table, whereas QEMU kept that same PTE in the TLB and did not re-walk. Ideally, the Sail model would specify a non-deterministic TLB that allows a wider range of observed behaviors, but fixing that is beyond the scope of this project.

The tests also uncovered a known shortcoming of the Sail model: the Sail model allows access only to HART 0's CLINT MMIO registers. When we ran the tests on a non-0 HART of the JH7110, we discovered that we could not match that behavior in our model. According to comments in the Sail source code, this is a known, deliberate limitation of the model.

14 Related work

OS kernel verification. Formal verification of OS kernels goes back to the security kernels of the 1970s: the MITRE PDP-11/45 kernel [72, 88], UCLA Secure Unix [103], PSOS [30], and KSOS [71]. These projects verified designs or partial implementations by hand or with early tools; Kit [9] was the first machine-checked proof of a kernel implementation, at the machine-code level of a small idealized processor without interrupts, devices, or multiple processors.

seL4 [54, 55] established that full functional correctness of a microkernel is feasible, by refinement from an abstract specification down to C and, later, to the binary [90]. The verified kernel is single-core, is mostly non-preemptible with interrupts handled by polling at a few points, treats address translation and the TLB through a discipline assumed of the hardware rather than a model of it, and rules DMA out of scope.

Verisoft [5] verified a microkernel on a simplified processor, and Verisoft XT [65] targeted Hyper-V, with the Hyper-V proofs left incomplete.

CertiKOS [39–41] verified a multi-core kernel with fine-grained locking by layered refinement, with interrupts and device drivers [18] modeled over an abstract machine; assembly code is handled through a verified compiler for a subset of x86, and the hardware page-table walker and DMA are not modeled.

Push-button verifiers such as Hyperkernel [74], Serval [75], and Nickel [93] avoid interactive proofs by restricting the kernel design (finite interfaces, no interrupts during system calls) so that symbolic evaluation of LLVM IR or RISC-V binaries is tractable. One consequence is that they do not support concurrency (or many other features present in xv6).

Other verified low-level systems include Komodo [34], verified in Dafny at the level of Arm assembly; SeKVM [66] and its relaxed-memory successor [98], which verify the core of KVM on Arm across cores; the Arm CCA firmware [67]; überSpark [102]; Ironclad [45]; and recent systems written in verified Rust such as VeriSMo [108] and Atmosphere [20].

Nearly all of this prior work reasons above the hardware: at the level of C, Rust, LLVM IR, or an idealized assembly language, over a machine model that abstracts away the page-table walker, the TLB, the precise semantics of traps and interrupts, and devices that access memory. Most were designed or redesigned for verification. MachCSL instead verifies an existing Unix-like kernel, against the Sail RISC-V semantics itself, with concurrent HARTs, hardware page-table walks, interrupts at any instruction, DMA, and power failures all part of the model rather than assumptions about it. Prior verified kernels also do not include a complete Unix-like environment: processes, fork/exec, pipes, file descriptors, and a crash-safe file system running inside the kernel.

Application-level theorems. Several projects have proven end-to-end theorems about an application's externally visible behavior down to the machine, as MachCSL does, but without a general-purpose OS kernel in between. The CLI stack [10] and Verisoft [1, 5] pursued pervasive verification from a gate-level processor through a compiler and a small kernel up to applications (in Verisoft's case, an email client), over simplified uniprocessor hardware. Ironclad Apps [45] verifies complete applications down to assembly, on top of the minimal Verve OS [106], for a single-threaded stack. Bedrock's web application case study [22] verifies a multithreaded application together with its cooperative thread library at the assembly level. CakeML's verified stack [69] and the Bedrock2 lightbulb and garage-door systems [28, 29] state their top-level theorems, much as MachCSL does, as predicates on the I/O trace of a verified processor running the application's machine code; in both cases the application runs on bare metal, with system calls or MMIO provided directly by a thin runtime. In contrast, MachCSL's application theorem is about user-level binaries (`init`, `sh`, `echo`, `cat`, `grep`, `sync`, and `seccomp`) running as separate processes on a multi-core Unix-like kernel: the theorem covers `fork`, `exec`, `wait`, file descriptors, loading binaries from a crash-safe file system across power cycles, preemptive scheduling across HARTs, and the kernel's handling of allocation failures, all of which show up in the legal console traces of Figure 2. MachCSL's kernel specifications are also generic, so that the application proof is layered on top of the kernel's proofs without modifying them (§7).

Several other kernel verification projects provide theorems about application execution but do not prove functional correctness of particular applications. For example, seL4's isolation proofs, as used on an autonomous helicopter in the HACMS program [56], show that untrusted components cannot interfere with critical ones. CertiKOS [39] proves contextual refinement, which says that any user program behaves the same over the kernel's implementation as over its abstract specification, but does not itself verify any user program against that specification. Mansky et al. [70] close this gap for a simple case, connecting VST proofs of C programs that perform console I/O to CertiKOS's specifications of the corresponding system calls, so that the program's guarantee is stated in terms of the OS-level I/O events; the processes there are sequential, and the connection is at the level of CompCert C and the CertiKOS abstract machine. Brun et al. [11] argue that enabling such application proofs should be a primary goal of OS verification.

File system verification. Verified file systems have mostly been developed as standalone artifacts. FSCQ [12, 17] introduced crash Hoare logic to specify and prove crash safety of a sequential file system, and DFSCQ [19] extended it to deferred durability and `fsync` semantics; DiskSec [47] added confidentiality. Yggdrasil [92] verified a file system by push-button symbolic execution against crash-refinement specifications, and Argosy [13] showed how to compose layers with recovery. Cogent [6] and Flashix [87] verified flash file systems, and VeriBetrKV [44] a crash-safe key-value store. Concurrent verified storage came with AtomFS [109], which proves linearizability of a concurrent (but not crash-safe) file system, and with Perennial [14], GoJournal [15], and DaisyNFS [16], which extend Iris with crash reasoning to verify concurrent, crash-safe journaling and an NFS server. Ntzik et al. [78] give a separation logic for fault-tolerant resources, and SibylFS [83] a tested specification of POSIX file system behavior.

The overall file system proof plan in MachCSL follows the approach taken in Perennial [14] and PoWER [64] in terms of introducing abstract ghost state to keep track of the durable on-disk state that survives reboots, as well as per-boot ghost state to keep track of in-memory state of file system transactions that are in progress but have not committed yet.

Prior work on verified file systems mostly assumes a disk with atomic sector writes behind a synchronous or abstract interface; the disk driver, DMA, and interrupts are out of scope, as are the process and file-descriptor machinery that share state with the file system.

In MachCSL the file system is verified inside the kernel, together with the disk driver and its DMA descriptors, the disk interrupt handler, and the disk model's per-sector writeback behavior, so that the crash invariant is established at the moment each sector reaches the disk rather than at an abstract disk interface. The proof also has to handle the interaction between the file system and the rest of a Unix kernel: files unlinked while open, file descriptors shared across `fork`, and `exec` reading a binary from the file system while replacing the process's address space.

Verification at the instruction level. Realistic ISA models have become the ground truth for machine-level reasoning. Sail [7] and its symbolic executor Isla [8] give executable, sub-instruction semantics for Arm and RISC-V, and the CHERI project proved architectural security properties (capability monotonicity) directly over Sail models of the whole instruction set [77]. Earlier ISA models in the same spirit include Fox and Myreen's HOL4 model of Armv7 [35] and the ACL2 x86isa model [38]; notably, Goel verified a system-mode program with paging enabled against x86isa [37], so hardware page-table walks were part of the semantics, although the reasoning was sequential and Hoare-style. These efforts prove properties *of the architecture*, or of small programs on it; MachCSL instead uses the Sail RISC-V model as the trusted semantics underneath a proof of an entire concurrent kernel.

Program logics over machine code predate Iris: Myreen and Gordon's Hoare logic for realistically modeled machine code [73], Jensen et al.'s separation logic for x86 in Coq [48], and Bedrock [21] all reason about assembly-level programs, but sequentially and over simplified machines. seL4's binary verification [90] and CakeML [61, 97] reach the binary by translation validation and verified compilation rather than by a logic over the ISA. The closest work to MachCSL in terms of connecting software proofs to a machine model is the Bedrock2 line [28, 29], which verifies software in separation logic against the riscv-coq ISA semantics, models an MMIO device as externally observable events, and connects down to a Kami-verified processor [23]. That work is single-HART and sequential, with no interrupts, page tables, or DMA; MachCSL's theorem is about a multi-HART kernel under all of these.

Concurrent separation logic at the machine level. Islaris [86] is the most direct precedent for MachCSL: an Iris program logic over Sail-derived instruction semantics for Arm and RISC-V, using Isla symbolic traces as the specification of each instruction. Islaris is sequential and its case studies are small (exception handlers and a few hundred instructions), with no page tables or devices; MachCSL can be seen as scaling this approach to an entire OS kernel with concurrency, interrupts, and devices. Kuru and Gordon's logic for x86-64 assembly [63], discussed below, is likewise an Iris logic at the instruction level, sequential, and focused on address translation.

Cerise [36] is an Iris logic at the assembly level of an idealized capability machine, with logical relations for reasoning about untrusted code, building on earlier capability-machine work [95]; later work adds MMIO devices and interrupts in a simplified form [101]. Cerise's treatment of unknown code running arbitrary instructions is analogous to MachCSL's invariant for user-mode execution (§6), although MachCSL relies on the page table and privilege mechanism of a real ISA rather than on capabilities.

Feng et al. [33] used CSL-style ownership transfer at interrupt enable and disable points at the assembly level, on an idealized machine with an idealized interrupt model, building on their earlier CSL for low-level code [32]. At the C level, CertiμC/OS [105] verifies a preemptive real-time kernel with CSL and interrupts, and CertiKOS [18] models devices as concurrent threads in a refinement framework. MachCSL differs from all of these in reasoning about the actual RISC-V interrupt, trap, and privilege machinery as specified by Sail, including the interaction of interrupts with page-table switches and context switches.

Hardware page tables and TLBs. Several program logics have made address translation first-class, but all of them are sequential. In the seL4 line, Kolanski and Klein's mapped separation logic [57–59] interprets assertions relative to a page-table root: a virtual points-to fact existentially quantifies the physical address that the virtual address translates to, and owns both the page-table entries traversed by the walk and the physical location itself, so that the frame rule survives page-table updates. Syeda and Klein [96] add an explicit TLB to the machine model and a logic that tracks addresses whose cached translations may be stale, and verify the Arm instruction sequence that installs a new page-table root and flushes the TLB. Kuru and Gordon [62, 63] build an Iris logic over a RISC-like fragment of

x86-64 assembly in which a virtual points-to fact is a ghost translation token, $va \hookrightarrow pa$, together with ownership of the physical location, while a separate per-address-space invariant owns the page-table memory that justifies the ghost map. This is the same decomposition that MachCSL uses for $va \mapsto_{\mathsf{vMem}} v$ (§5); in both cases it allows the page tables to be modified without collecting the virtual points-to facts that depend on them. On top of it, Kuru and Gordon introduce a hybrid-logic modality $[r]P$, stating that P holds in the address space rooted at r, and the proof rule for writing the page-table root register moves assertions into and out of this modality. With it they verify a context switch that changes address spaces, and a software page-table walk and page-mapping routine from an xv6-derived kernel, including the kernel's use of constant-offset (identity) mappings to reach page-table pages from their physical addresses. Their machine model has a single core and no TLB, and page tables are modified only by software.

Kernel-scale proofs have modeled address translation more coarsely. Verisoft verified a paging mechanism against a simple hardware MMU [2, 4], and Verisoft XT modeled the TLB walk of a hypervisor's shadow page tables as a separate concurrent thread whose memory accesses are governed by VCC's ownership discipline [3, 25]; this is the closest prior treatment of a hardware page-table walk racing with kernel code, though on a simplified x86 model rather than a machine-checked ISA semantics. Verified hypervisors reason about page tables and TLB invalidation across cores by refinement: SeKVM [66] and its extension to Arm relaxed memory [98], Komodo [34], and the PROSPER line, which verifies MMU virtualization for Linux on Arm against an Armv7 ISA model in HOL4 [42, 76]. Simner et al. [94] give a relaxed virtual-memory model for Armv8-A in which concurrent hardware walks are first-class, but as a model of the architecture rather than a proof of software over it. MachCSL treats the RISC-V page-table walker and its A/D-bit writeback as concurrent hardware activity whose accesses to page-table memory are governed by CSL invariants, on the same footing as accesses by other HARTs, and the TLB is part of the model rather than a discipline assumed of the hardware. MachCSL does not need a modality for other address spaces: all of xv6's kernel code runs under one kernel page table shared by every HART, with the translation tiers of §6 distinguishing the boot-time regimes, and the switch to a user page table is reasoned about through the page-table-switching window in the trampoline and the invariant for user-mode execution.

Shared memory and memory models. Formal models of the shared-memory behavior of multiprocessors are well established: x86-TSO [89] is the canonical store-buffer model, and RISC-V's architected memory model, RVWMO [84], has formal axiomatic and operational definitions, including an operational model in Coq [81] in the style of the promising semantics [52]. RISC-V also architects TSO itself, as the Ztso extension [84], a legal strengthening of RVWMO.

Program logics for weak memory have so far been exercised only on small programs. Ridge [82] verified a spinlock with rely-guarantee reasoning over x86-TSO. iCAP-TSO [91] is the direct precedent for MachCSL's memory model: a higher-order separation logic over an operational TSO semantics, whose "fiction of sequential consistency" lets well-synchronized code be verified with SC-style rules while the racy code that implements synchronization is verified against the store buffers directly. Its programs are written in an idealized language, and its case studies are locks and small data structures. GPS and its Iris-based successors target language-level models such as C11 [27, 100], and AxSL [43] brings Iris to the relaxed Arm-A memory model at the ISA level, but for litmus-test-sized programs. MachCSL's views are inspired by views from GPFSL [27]. One difference in the way the xv6 kernel proofs use views is the idea of suspended views, which are used to reason about context switching and migrating threads between cores.

No prior kernel-scale proof reasons in a weak-memory logic. Kernel verification under weak memory has instead gone through reduction theorems that recover sequential consistency: Cohen and Schirmer [24] proved that code obeying a store-buffer flush discipline exhibits only SC behaviors on TSO, developed for the (unfinished) Hyper-V verification at the C level, and SeKVM [98] was verified over Arm relaxed memory by proving that the kernel follows a synchronization discipline under which its relaxed executions reduce to SC ones. Armada [68] verifies concurrent data structures by TSO-aware refinement, and VSync [79] model-checks kernel synchronization primitives under weak-memory models rather than proving them deductively. The reduction approach breaks down where a kernel is intentionally racy: e.g., xv6 uses flag variables in shared memory for lock-free coordination. MachCSL instead verifies xv6 directly in a program logic over a weak model of shared memory: well-synchronized code is verified with SC-style reasoning, and xv6's racy code against the TSO-with-load-load-reordering semantics itself.

Devices and DMA. To our knowledge, no prior machine-checked kernel proof reasons about a DMA-capable device model running concurrently with the CPU. seL4 assumes DMA away (devices are trusted or confined by an IOMMU); CertiKOS's device objects [18] are interrupt-driven and do not access memory; Verisoft's device models [46] use programmed I/O; and the Bedrock2 lightbulb device [28] is a stream of MMIO events. Penninckx et al. [80] express device I/O in separation logic, but at the C level with abstract I/O actions. The xv6 proof transfers ownership of DMA

buffers between the kernel and the disk controller's model, so that the proof accounts for every byte the device reads or writes and for the moment each sector reaches the disk.

15 Change log

Changes since arXiv:2609.04043v1 [51]. Added application-level correctness theorem (§2). Added verification of user-level applications (§7). Added non-coherent instruction cache model. Weakened shared-memory model to allow load-load reordering. Added subsections §6.15 and §10.4. Improved CSL explanation (§4). Various clarifications.

Changes since arXiv:2609.04043v2. Added a more sophisticated top-level theorem that allows redirection to files, pipes, and sandboxing of unverified binaries using `seccomp` (§8). Ported the development from Rocq to Lean. Found a bug in `grep` and a bug in Sail's Lean extraction. Extended conformance testing to the CVA6 implementation in Verilator.

16 Acknowledgments

Thanks to Jason Gross and Tej Chajed for inspiring us to work on agent-assisted verification. Thanks to Baltasar Dinis, Yun-Sheng Chang, Colin Gordon, Gernot Heiser, Ismail Kuru, Jay Lorch, Natalie Neamtu, and Deian Stefan for feedback that helped improve this paper. This work was supported by NSF award CNS-2225441 and by gifts from Amazon and Google.

References

[1] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 209–224, Toronto, Canada, Oct. 2008.

[2] E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 109–123, Budapest, Hungary, Mar.–Apr. 2008.

[3] E. Alkassar, E. Cohen, M. A. Hillebrand, M. Kovalev, and W. J. Paul. Verifying shadow page table algorithms. In *Proceedings of the 10th Formal Methods in Computer-Aided Design (FMCAD)*, pages 267–270, Lugano, Switzerland, Oct. 2010.

[4] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Proceedings of the 3rd Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 40–54, Edinburgh, United Kingdom, Aug. 2010.

[5] E. Alkassar, W. J. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In *Proceedings of the 3rd Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 71–85, Edinburgh, United Kingdom, Aug. 2010.

[6] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, Atlanta, GA, Apr. 2016.

[7] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, Jan. 2019.

[8] A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV)*, pages 303–316, Los Angeles, CA, July 2021.

[9] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11): 1382–1396, Nov. 1989.

[10] W. R. Bevier, W. A. Hunt, Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, Dec. 1989.

[11] M. Brun, R. Achermann, T. Chajed, J. Howell, G. Zellweger, and A. Lattuada. Beyond isolation: OS verification as a foundation for correct applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS)*, Providence, RI, June 2023.

[12] T. Chajed, H. Chen, A. Chlipala, M. F. Kaashoek, N. Zeldovich, and D. Ziegler. Certifying a file system using Crash Hoare Logic: Correctness in the presence of crashes. *Communications of the ACM*, 60(4):75–84, Apr. 2017.

[13] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1037–1051, Phoenix, AZ, June 2019.

[14] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, Huntsville, Ontario, Canada, Oct. 2019.

[15] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–439, Virtual conference, July 2021.

[16] T. Chajed, J. Tassarotti, M. Theng, M. F. Kaashoek, and N. Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 447–463, Carlsbad, CA, July 2022.

[17] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, Oct. 2015.

[18] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 431–447, Santa Barbara, CA, June 2016.

[19] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, Oct. 2017.

[20] X. Chen, Z. Li, J. Zhang, V. Narayanan, and A. Burtsev. Atmosphere: Practical verified kernels with Rust and Verus. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, Seoul, South Korea, Oct. 2025.

[21] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 391–402, Boston, MA, Sept. 2013.

[22] A. Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 609–622, Mumbai, India, Jan. 2015.

[23] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Oxford, United Kingdom, Sept. 2017.

[24] E. Cohen and N. Schirmer. From total store order to sequential consistency: A practical reduction theorem. In *Proceedings of the 1st International Conference on Interactive Theorem Proving*, Edinburgh, United Kingdom, July 2010.

[25] E. Cohen, W. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In *Proceedings of the 39th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 1–27, Špindlerův Mlýn, Czech Republic, Jan. 2013.

[26] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2026. https://github.com/mit-pdos/xv6-riscv.

[27] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer. RustBelt meets relaxed memory. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, LA, Jan. 2020.

[28] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Virtual conference, June 2021.

[29] A. Erbsen, J. Philipoom, D. Jamner, A. Lin, S. Gruetter, C. Pit-Claudel, and A. Chlipala. Foundational integration verification of a cryptographic server. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Copenhagen, Denmark, June 2024.

[30] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the 1979 National Computer Conference*, pages 329–334, 1979.

[31] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–65, West Lafayette, IN, Nov. 1977.

[32] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proceedings of the 16th European Symposium on Programming (ESOP)*, pages 173–188, Braga, Portugal, Mar. 2007.

[33] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 170–182, Tucson, AZ, June 2008.

[34] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 287–305, Shanghai, China, Oct. 2017.

[35] A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the 1st International Conference on Interactive Theorem Proving*, pages 243–258, Edinburgh, United Kingdom, July 2010.

[36] A. L. Georges, A. Guéneau, T. Van Strydonck, A. Timany, A. Trieu, D. Devriese, and L. Birkedal. Cerise: Program verification on a capability machine in the presence of untrusted code. *Journal of the ACM*, 71(1):3:1–3:59, Feb. 2024.

[37] S. Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model.* PhD thesis, The University of Texas at Austin, 2016.

[38] S. Goel, W. A. Hunt, Jr., and M. Kaufmann. Engineering a formal, executable x86 ISA simulator for software verification. *Provably Correct Systems*, pages 173–209, 2017.

[39] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 595–608, Mumbai, India, Jan. 2015.

[40] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, Nov. 2016.

[41] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 646–661, Philadelphia, PA, June 2018.

[42] R. Guanciale, H. Nemati, M. Dam, and C. Baumann. Provably secure memory isolation for Linux on ARM. *Journal of Computer Security*, 24(6):793–837, 2016.

[43] A. Hammond, Z. Liu, T. Pérami, P. Sewell, L. Birkedal, and J. Pichon-Pharabod. An axiomatic basis for computer programming on the relaxed Arm-A architecture: The AxSL logic. In *Proceedings of the 51st ACM Symposium on Principles of Programming Languages (POPL)*, London, United Kingdom, Jan. 2024.

[44] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–115, Virtual conference, Nov. 2020.

[45] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, Oct. 2014.

[46] M. Hillebrand, T. In der Rieden, and W. Paul. Dealing with I/O devices in the context of pervasive system verification. In *Proceedings of the 23rd IEEE International Conference on Computer Design (ICCD)*, pages 309–316, San Jose, CA, Oct. 2005.

[47] A. İleri, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Proving confidentiality in a file system using DiskSec. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 323–338, Carlsbad, CA, Oct. 2018.

[48] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)*, pages 301–314, Rome, Italy, Jan. 2013.

[49] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, Jan. 2015.

[50] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[51] M. F. Kaashoek and N. Zeldovich. Extending concurrent separation logic to the hardware level to verify the xv6 OS kernel on RISC-V with AI agents. arXiv:2609.04043 [cs.LO], Sept. 2026. Available at https://arxiv.org/abs/2609.04043.

[52] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 175–189, Paris, France, Jan. 2017.

[53] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings of the 11th IEEE Symposium on Security and Privacy*, pages 2–19, Oakland, CA, May 1990.

[54] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, Oct. 2009.

[55] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–70, Feb. 2014.

[56] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. Murray, and G. Heiser. Formally verified software in the real world. *Communications of the ACM*, 61(10):68–77, Oct. 2018.

[57] R. Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, University of New South Wales, July 2011.

[58] R. Kolanski and G. Klein. Mapped separation logic. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 15–29, Toronto, Canada, Oct. 2008.

[59] R. Kolanski and G. Klein. Types, maps and separation logic. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 276–292, Munich, Germany, Aug. 2009.

[60] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *Proceedings of the 26th European Symposium on Programming (ESOP)*, pages 696–723, Uppsala, Sweden, Apr. 2017.

[61] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL)*, pages 179–192, San Diego, CA, Jan. 2014.

[62] I. Kuru. *Modal Abstractions for Operating System Kernels*. PhD thesis, Drexel University, Mar. 2025.

[63] I. Kuru and C. S. Gordon. Modal abstractions for virtualizing memory addresses. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):2338–2366, Oct. 2025.

[64] H. LeBlanc, J. R. Lorch, C. Hawblitzel, C. Huang, Y. Tao, N. Zeldovich, and V. Chidambaram. PoWER never corrupts: Tool-agnostic verification of crash consistency and corruption detection. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2025.

[65] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 16th International Symposium on Formal Methods (FM)*, pages 806–809, Eindhoven, The Netherlands, Nov. 2009.

[66] S. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *Proceedings of the 30th USENIX Security Symposium*, Vancouver, Canada, Aug. 2021.

[67] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. Design and verification of the Arm confidential compute architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022.

[68] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210, London, United Kingdom, June 2020.

[69] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1041–1053, Phoenix, AZ, June 2019.

[70] W. Mansky, W. Honoré, and A. W. Appel. Connecting higher-order separation logic to a first-order outside world. In *Proceedings of the 29th European Symposium on Programming (ESOP)*, Dublin, Ireland, Apr. 2020.

[71] E. J. McCauley and P. J. Drongowski. KSOS—the design of a secure operating system. In *Proceedings of the 1979 National Computer Conference*, pages 345–354, 1979.

[72] J. K. Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, May 1976.

[73] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 568–582, Braga, Portugal, Mar.–Apr. 2007.

[74] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 252–269, Shanghai, China, Oct. 2017.

[75] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 225–242, Huntsville, Ontario, Canada, Oct. 2019.

[76] H. Nemati, R. Guanciale, and M. Dam. Trustworthy virtualization of the ARMv7 memory subsystem. In *Proceedings of the 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 578–589, Pec pod Sněžkou, Czech Republic, Jan. 2015.

[77] K. Nienhuis, A. Joannou, T. Bauereiss, A. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 1003–1020, San Francisco, CA, May 2020.

[78] G. Ntzik, P. da Rocha Pinto, and P. Gardner. Fault-tolerant resource reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 169–188, Pohang, South Korea, Nov.–Dec. 2015.

[79] J. Oberhauser, R. L. de Lima Chehab, D. Behrens, M. Fu, A. Paolillo, L. Oberhauser, K. Bhat, Y. Wen, H. Chen, J. Kim, and V. Vafeiadis. VSync: Push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual conference, Apr. 2021.

[80] W. Penninckx, B. Jacobs, and F. Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In *Proceedings of the 24th European Symposium on Programming (ESOP)*, pages 158–182, London, United Kingdom, Apr. 2015.

[81] C. Pulte, J. Pichon-Pharabod, J. Kang, S.-H. Lee, and C.-K. Hur. Promising-ARM/RISC-V: A simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–15, Phoenix, AZ, June 2019.

[82] T. Ridge. A rely-guarantee proof system for x86-TSO. In *Proceedings of the 3rd Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, Edinburgh, United Kingdom, Aug. 2010.

[83] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 38–53, Monterey, CA, Oct. 2015.

[84] RISC-V International. The RISC-V instruction set manual, volume I: Unprivileged architecture. https://riscv.org/specifications/ratified/, 2024.

[85] RISC-V International. Formal specification of the RISC-V ISA. https://github.com/riscv/sail-riscv, 2026.

[86] M. Sammler, A. Hammond, R. Lepigre, B. Campbell, J. Pichon-Pharabod, D. Dreyer, D. Garg, and P. Sewell. Islaris: Verification of machine code against authoritative ISA semantics. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 825–840, San Diego, CA, June 2022.

[87] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proceedings of the ABZ Conference*, pages 9–24, Toulouse, France, June 2014.

[88] W. L. Schiller. The design and specification of a security kernel for the PDP-11/45. Technical Report MTR-2934, MITRE Corp., Bedford, MA, Mar. 1975.

[89] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[90] T. Sewell, M. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, Seattle, WA, June 2013.

[91] F. Sieczkowski, K. Svendsen, L. Birkedal, and J. Pichon-Pharabod. A separation logic for fictional sequential consistency. In *Proceedings of the 24th European Symposium on Programming (ESOP)*, London, United Kingdom, Apr. 2015.

[92] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, Nov. 2016.

[93] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 287–306, Carlsbad, CA, Oct. 2018.

[94] B. Simner, A. Armstrong, J. Pichon-Pharabod, C. Pulte, R. Grisenthwaite, and P. Sewell. Relaxed virtual memory in Armv8-A. In *Proceedings of the 31st European Symposium on Programming (ESOP)*, pages 143–173, Munich, Germany, Apr. 2022.

[95] L. Skorstengaard, D. Devriese, and L. Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, Jan. 2019.

[96] H. T. Syeda and G. Klein. Formal reasoning under cached address translation. *Journal of Automated Reasoning*, 64:911–945, 2020.

[97] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 60–73, Nara, Japan, Sept. 2016.

[98] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu. Formal verification of a multiprocessor hypervisor on Arm relaxed memory hardware. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 866–881, Virtual conference, Oct. 2021.

[99] The Coq Development Team. *The Coq Proof Assistant, version 8.17.1*, June 2023. URL https://doi.org/10.5281/zenodo.8161141.

[100] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 29th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 691–707, Portland, OR, Oct. 2014.

[101] T. Van Strydonck, A. L. Georges, A. Guéneau, A. Trieu, A. Timany, F. Piessens, L. Birkedal, and D. Devriese. Proving full-system security properties under multiple attacker models on capability machines. In *Proceedings of the 35th IEEE Computer Security Foundations Symposium (CSF)*, pages 80–95, Haifa, Israel, Aug. 2022.

[102] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überSpark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *Proceedings of the 25th USENIX Security Symposium*, pages 87–104, Austin, TX, Aug. 2016.

[103] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, Feb. 1980.

[104] L. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: Representing recursive and impure programs in Coq. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*, pages 51:1–51:32, New Orleans, LA, Jan. 2020.

[105] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive OS kernels. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*, pages 59–79, Toronto, Canada, July 2016.

[106] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, Toronto, Canada, June 2010.

[107] N. Zeldovich. rocq-warm. https://github.com/zeldovich/rocq-warm, Sept. 2026.

[108] Z. Zhou, Anjali, W. Chen, S. Gong, C. Hawblitzel, and W. Cui. VeriSMo: A verified security module for confidential VMs. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024.

[109] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen. Using concurrent relational logic with helper for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, Oct. 2019.