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

September 3, 2026

Abstract

MachCSL is a framework for verifying systems 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 (6,593 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 nine bugs in the xv6 implementation, as well as one bug in the Sail RISC-V semantics. The verification effort took us 77 days, including the time to develop the MachCSL framework.

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 CPUs; interrupts taken during user-space execution and even during kernel execution itself, either from devices or timer interrupts; 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 [17, 26, 27, 30, 36, 48, 49, 56, 57, 59, 60, 62, 76, 79, 82, 92, 95].

The contribution of this work is a machine-checked proof of the xv6 kernel [22] directly on a low-level model of the RISC-V hardware [73], 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 CPUs; 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; handles concurrent DMA from devices; and includes a Unix file system and system call interface. xv6 implements the file system and system calls with fine-grained locking, which introduces 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 [73], 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,

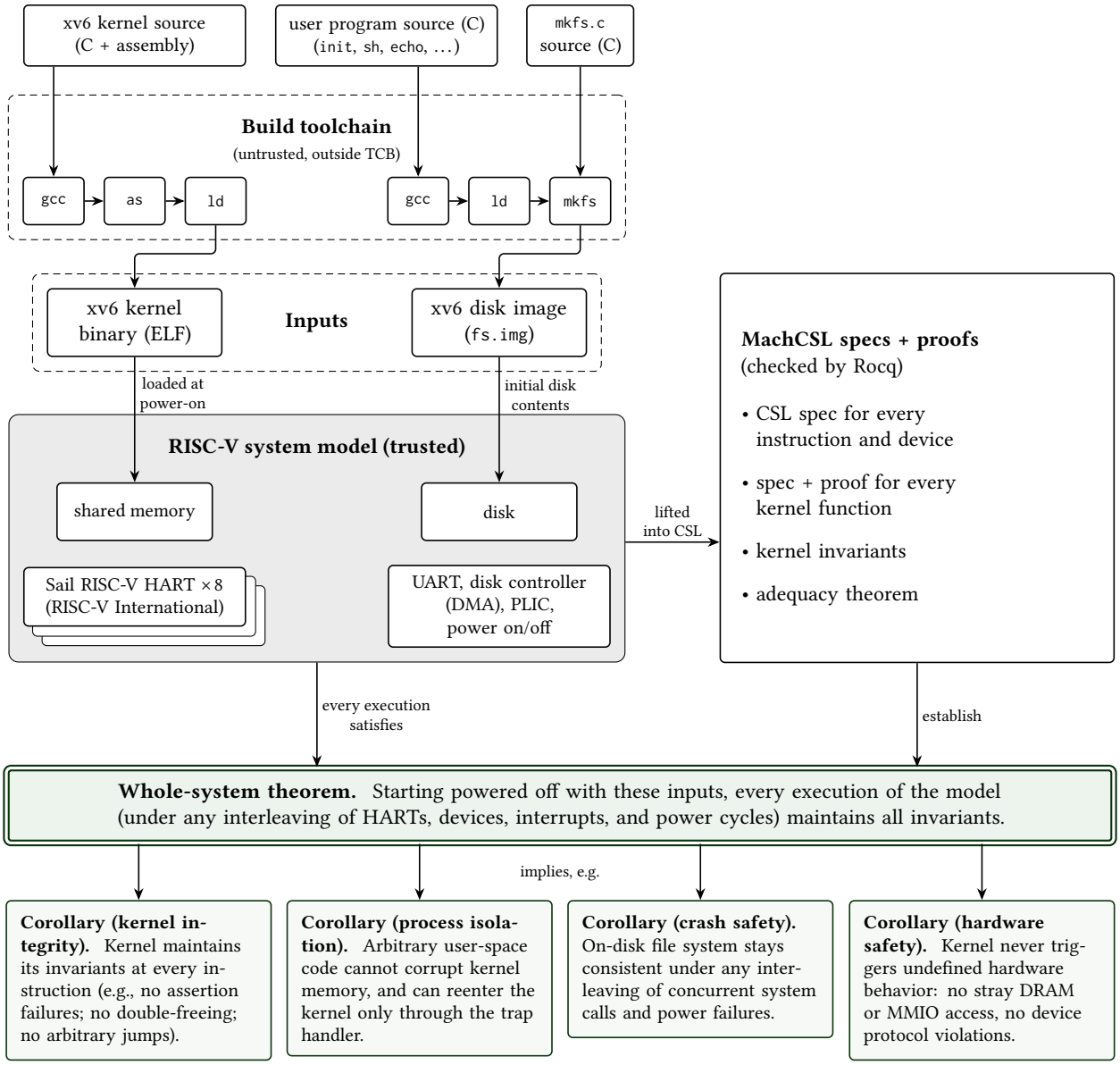

Figure 1: An overview of the proof approach for verifying the xv6 kernel.

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 whole-system 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 whole-system 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 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 [45, 46, 52] 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 CPU, 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 make extensive use of AI agents (Claude Code) to help us write the proofs and intermediate specs. One pleasant benefit of using agents for proofs is that we don't have to inspect them because they are machine-checked by Rocq [88], which will validate that the proofs imply the desired top-level theorem, such as our whole-system theorem.

Our whole-system theorem and verification project have some limitations. First, our proofs cover safety properties (e.g., the kernel does not crash), but do not yet cover liveness properties (e.g., every process will keep making progress), 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 whole-system theorem is subject to the assumption that our 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 whole-system theorem applies to our model of TSO shared memory. Finally, the verification relies on the whole-system theorem being meaningful: what we prove is the precise whole-system theorem statement, rather than the more vague "xv6 is correct or bug-free".

The overall proof of the whole-system theorem about the xv6 kernel comprises about 1.3 million lines of Rocq code; the xv6 kernel itself, which we verified, is 6,593 lines of C and assembly, which compile down to 8,607 RISC-V instructions. The proof effort took 77 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 several bugs in xv6, including bugs that would crash the kernel, as well as a bug in the Sail RISC-V model (§8). Other than fixing these kernel 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 that was not designed or implemented with verification in mind. The whole-system theorem implies that the xv6 kernel correctly isolates user process code, correctly recovers the file system after crashes, and never fails any of its internal consistency assertions.

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

2 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 CPU's execution. Specifically, MachCSL uses the formal specification of the RISC-V ISA written in Sail, as adopted by RISC-V International [73]. 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, increment the clock cycle, check pending interrupts, vector to the trap handler, etc. The 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 §8).

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 a UART 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 captures TSO semantics, which allows CPUs to implement write-buffering. 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's visible to the core for purposes of memory reads. Thus, when a core executes a memory read, it gets the last write event to that address as of the core's timestamp. The one exception to this rule is that, under TSO, cores see their own writes (i.e., they check their own store buffer). Thus, in our model, if the latest write to the address being read came from the same core, then a read will return it. 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. Fence instructions advance the core's timestamp to its own latest write (to any address). Our model currently does not have a separate instruction cache (RISC-V allows for a non-cache-coherent icache).

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 some number of concurrent HARTs, representing all of the cores. The HARTs start running at the initial PC as specified by RISC-V (0x80000000), starting 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 contents of the disk is 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).

3 MachCSL by example

To explain how MachCSL uses concurrent separation logic (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, along with the rules that CSL provides for reasoning about such code. MachCSL in particular builds on top of the Iris concurrent separation logic [45, 46, 52], as mechanized in Coq/Rocq [88]. A key benefit of CSL is that it provides powerful abstractions for reasoning about execution of concurrent code, in a way that allows for largely independent proofs of different concurrently executing threads, even when the threads share state. The C code of the example spinlock implementation that we will consider is shown in Figure 2. 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 3. 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.

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 for a.

```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 2: 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 3: Compiled assembly of the spinlock from Figure 2.

Much as in 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 4 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`. kernel_text is a requirement that the kernel's text section (instructions) is present in memory (which happens to include the code of `acquire` and `release`). pc_is `acquire` is a requirement that the `pc` register contains the address of `acquire`, which is where `acquire`'s first instruction lives, as shown in Figure 3. `a0` $\mapsto_{\text{Reg}}$ l requires that the `a0` register—the first argument in the RISC-V calling convention—contains the address of the lock, l. The next two preconditions require passing ownership of both the `a4` and `a5` registers, containing any value; `acquire` doesn't 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. We will discuss the is_lock predicate, and the parenthesized continuation, later in this section.

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 post-conditions associated with some function. In this style, one might expect that there is a clear pre- and post-condition associated with running `acquire`.

Having explicit pre- and post-conditions 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

kernel_text −∗
pc_is acquire −∗
a0 $\mapsto_{\text{Reg}}$ l −∗
∃ v_4. a4 $\mapsto_{\text{Reg}}$ v_4 −∗
∃ v_5. a5 $\mapsto_{\text{Reg}}$ v_5 −∗
ra $\mapsto_{\text{Reg}}$ r −∗
is_lock l P −∗
(pc_is r −∗
  a0 $\mapsto_{\text{Reg}}$ l −∗
  ∃ v_4. a4 $\mapsto_{\text{Reg}}$ v_4 −∗
  ∃ v_5. a5 $\mapsto_{\text{Reg}}$ v_5 −∗
  ra $\mapsto_{\text{Reg}}$ r −∗
  P −∗
  wp CpuLoop) −∗
wp CpuLoop

(a) Specification for `acquire`.

kernel_text −∗
pc_is release −∗
a0 $\mapsto_{\text{Reg}}$ l −∗
ra $\mapsto_{\text{Reg}}$ r −∗
is_lock l P −∗
P −∗
(pc_is r −∗
  a0 $\mapsto_{\text{Reg}}$ l −∗
  ra $\mapsto_{\text{Reg}}$ r −∗
  wp CpuLoop) −∗
wp CpuLoop

(b) Specification for `release`.

Figure 4: Specifications for the example spinlock in Figure 3.

is not how hardware semantics are defined: there is no notion of functions, or returning to a caller, and there isn't 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 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's safe to run the CPU in this way using the resource wp CpuLoop.[1] 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: 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 xv6 kernel's implementation requires establishing wp CpuLoop at the initial CPU starting configuration—that is, it's safe to enter this "infinite loop" starting from whatever state the CPU happens to have when it first powers up.

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's 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.

[1] "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 turns out to allow the MachCSL proof to conclude a strong adequacy theorem that implies strong corollaries about the xv6 kernel, as illustrated in Figure 1.

To enable this decomposition, most specifications in MachCSL are formulated in terms of *continuations*. For example, the specification of `acquire`, shown in Figure 4, says that it can prove the safety of wp CpuLoop, 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's 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, the safety of context-switching to another kernel thread, etc.

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's 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. The specification for `release` does the opposite operation with P: it's 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.

Instruction specs. To prove the correctness of the spinlock implementation, a proof in MachCSL must reason about the individual instructions that comprise the implementation, chaining them together to conclude the specification of the entire `acquire` or `release` function. For example, Figure 5 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.

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

(a) Specification for `LI`.

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

(b) Specification for `RET`.

$$
\begin{array}{l}
\text{pc\_is } i \mathrel{-\!\!*} \\
\text{instr } i \ \ (\texttt{AMOSWAP}\ r_v, r_v, (r_a)) \mathrel{-\!\!*} \\
r_a \mapsto_{\text{Reg}} a \mathrel{-\!\!*} \\
r_v \mapsto_{\text{Reg}} v \mathrel{-\!\!*} \\
\dot{\Rightarrow}(\exists\, x.\ \ a \mapsto_{\text{Mem}} x * (a \mapsto_{\text{Mem}} v \mathrel{-\!\!*} \dot{\Rightarrow} Q\ v\ x)) \mathrel{-\!\!*} \\
(\ \text{pc\_is } (i+4) \mathrel{-\!\!*} \\
\quad r_a \mapsto_{\text{Reg}} a \mathrel{-\!\!*} \\
\quad r_v \mapsto_{\text{Reg}} x \mathrel{-\!\!*} \\
\quad Q\ v\ x \mathrel{-\!\!*} \\
\quad \text{wp CpuLoop}) \mathrel{-\!\!*} \\
\text{wp CpuLoop}
\end{array}
$$

(c) Specification for `AMOSWAP`.

Figure 5: Specifications for individual RISC-V instructions.

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 5. The `acquire` proof holds the resource pc_is `acquire`, as well as the resource $\texttt{a4} \mapsto_{\text{Reg}} v_4$, 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 was omitted from the `acquire` specification for simplicity. 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 pc_is (`acquire` + 2) as well as $\texttt{a4} \mapsto_{\text{Reg}} 1$. This allows the proof of `acquire` to keep stepping through its instructions until it hits the return instruction, `RET`. The specification of `RET` requires proving a wp CpuLoop for pc_is 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.

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 of callers 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:

$$\mathsf{kernel\_text} * \mathsf{pc\_is}\ \texttt{acquire} * \texttt{a0} \mapsto_{\mathsf{Reg}} l * \texttt{a4} \mapsto_{\mathsf{Reg}} v_4 * \texttt{a5} \mapsto_{\mathsf{Reg}} v_5 * \texttt{ra} \mapsto_{\mathsf{Reg}} r * \mathsf{is\_lock}\ l\ P$$

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 $\mathsf{instr}\ i\ (\texttt{LI}\ r, x)$ fact from kernel_text, with $i = \texttt{acquire}$, $r = \texttt{a4}$, and $x = 1$. The proof then passes ownership of $\mathsf{instr}\ i\ \ (\texttt{LI}\ r, x)$, $\mathsf{pc\_is}\ i$, and $\texttt{a4} \mapsto_{\mathsf{Reg}} v_4$ 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 $\mathsf{pc\_is}\ (i + 2)$ and $\texttt{a4} \mapsto_{\mathsf{Reg}} 1$, 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.

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. 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 6. 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's no P in the lock invariant (it has been handed out to the acquiring thread).

$$\mathsf{is\_lock}\ l\ P \triangleq \exists\, v.\ \ l \mapsto_{\mathsf{Mem}} v * ((v = 0 * P) \vee (v = 1))$$

Figure 6: 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 5(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 $\dot{\Rightarrow}\ldots$ resource, as we will now explain.

The $\dot{\Rightarrow}$ notation encodes a proof callback (in Iris, this object is called an update modality or fancy-update modality). Specifically, $\dot{\Rightarrow} A$ is a resource that allows the holder to "invoke" the callback and receive ownership of A. The precondition of `AMOSWAP` uses $\dot{\Rightarrow}$ to encode the fact that, at some instant during `AMOSWAP`'s execution, the caller will be required to give ownership of $a \mapsto_{\mathsf{Mem}} x$, for some value x, and also give another $\dot{\Rightarrow}$ resource: a proof callback that, given $a \mapsto_{\mathsf{Mem}} v$, 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 $\dot{\Rightarrow}$ statements, requiring them to be executed together at the same point in the proof (invoking the outer $\dot{\Rightarrow}$ obligates the invoker to also invoke the inner $\dot{\Rightarrow}$ 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 $\dot{\Rightarrow}$ corresponds to allowing the caller to open the invariant, and the inner $\dot{\Rightarrow}$ 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 $\dot{\Rightarrow}$ callback opens the lock invariant

and gives up $a \mapsto_{\mathsf{Mem}} x$, and the inner callback receives $a \mapsto_{\mathsf{Mem}} v$ 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.

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 page table and a TLB representing a logical mapping from virtual to physical addresses (abstracting away the 3-level 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. 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, etc. 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.

4 CSL for RISC-V

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

4.1 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 system 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 $\texttt{tlb} \mapsto_{\mathsf{Reg}} \mathit{tlb\text{-}entries}$. 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 sub-divides 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 with TSO semantics, 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 doesn't 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, disk, and PLIC in xv6). 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 §3 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 $\dot{\Rrightarrow}$ to open the device invariant and describe what effect that particular MMIO access has on the device state.

4.2 Execution

At the whole-instruction level, execution is captured in MachCSL by the wp CpuLoop resource, representing the fact that the HART can execute for a whole RISC-V cycle. The CpuLoop operation is indexed by a HART ID, which determines the set of registers that are used in executing that cycle; much like register points-to facts, we 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.

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) {Φ}, 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) {Φ} 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, each cycle's execution is precisely defined by Sail's model. This is in contrast to executing a RISC-V instruction, which is determined by dynamic state, such as page tables, memory contents, interrupts, etc. 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 forever 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.

To reason about the steps of the Sail model, MachCSL uses standard sequencing combinators. Proving wp Sail(a; b) {Φ} requires proving wp Sail(a) {wp Sail(b) {Φ}}, meaning that, to prove that Φ holds after running a; b, it suffices to prove that, after running a, there's a wp Sail(b) {Φ} 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`) {Φ}, with a Φ that specifies how address translation works (i.e., the result of 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, incrementing the next PC, etc.

Each individual wp CpuLoop specification, such as the ones shown in Figure 5, 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 RMW 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, as well as 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) {Φ} 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.

4.3 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 proves that, for any execution of the system according to the operational semantics of our system model, as described in §2, 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's 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 contain 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 theorem proves that these initial bytes produced by `mkfs` satisfy the file system's invariant, bootstrapping the inductive reasoning across arbitrarily many power cycles.

5 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.

5.1 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 core (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, the set of general-purpose registers on the current HART.

Every developer-visible instruction specification in MachCSL takes `sie_cap_gpr` as a precondition and returns it back 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 $r \mapsto_{\text{Reg}} v$ 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 core, 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 $\exists\, h.$ `stvec` $\mapsto_{\text{Reg}} h *$ (pc_is $h \mathrel{-\!\!*} \ldots \mathrel{-\!\!*}$ wp CpuLoop). This says that `stvec` has some address 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.

5.2 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 `tlb` register. 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 on that HART. At the same time, the kernel uses a single page table shared across all cores, so the A/D writeback on one core can modify the page table that another core 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 system 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 address pointer ($va \mapsto_{\text{vMem}} v$) 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 always valid, both with and without a 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 must 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 contains a page of memory 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).

5.3 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 7 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 7: 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's some tuple *c* of registers saved at *ptr*, and, if the CPU restores the callee-saved registers from *c*, it can resume execution at *c.PC*.

The spec for `swtch` itself, then, 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's a valid context at the new pointer (second argument,

```
own_context ptr c ≜
  ptr + 0  ↦vMem  c.RA ∗
  ptr + 8  ↦vMem  c.SP ∗
  ptr + 16 ↦vMem  c.S0 ∗
  ...
  ptr + 104 ↦vMem  c.S11

valid_context ptr ≜
  ∃ c. own_context ptr c ∗
  ∀ m. sie_cap_gpr m ... −∗
       callee_saved m = (c.S0, ..., c.S11) −∗
       pc_is c.RA −∗
       wp CpuLoop

kernel_text −∗
pc_is swtch −∗
sie_cap_gpr m ... −∗
∃ c. own_context m[a0] c −∗
valid_context m[a1] −∗
(∀ m'. pc_is m[ra] −∗
       sie_cap_gpr m' ... −∗
       callee_saved m = callee_saved m' −∗
       ∃ c. own_context m[a0] c −∗
       valid_context m[a1] −∗
       wp CpuLoop) −∗
wp CpuLoop
```

Figure 8: Simplified specification for `swtch`.

passed in `a1`), and the caller can prove that it's safe to resume execution at the return address with callee-saved registers, then the caller can provably invoke `swtch`. Not shown in this spec is the implicit HART ID argument that indicates that `swtch` might return on another HART. Also 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.

5.4 Locks

The xv6 kernel uses spinlocks to coordinate access to shared data structures between concurrent threads. §3 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.

Disabling interrupts for spinlocks. xv6 couples spinlocks with interrupt handling, because it's important that a CPU holding a spinlock is not 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-enabled state, 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 an argument to `sie_cap_gpr` that represents 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-enabled flag 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-enabled flag 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 core on which they are acquired. This is represented by the `acquire` function returning an additional lock_held resource that's 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's 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 cores. 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's about to acquire is ranked higher than any other lock in the currently held lock set. This ensures both that the lock it is about to acquire cannot already be held by the same core (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's 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's 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 sleeplock's acquire requires that the caller have an empty lock 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 its in-memory refcount is now zero. 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 resource proving that the refcount is non-zero). Any acquired sleeplock holds this resource until the sleeplock is released. When the caller knows that the refcount is zero, 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.

5.5 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's 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's 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 duplicable, 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.

5.6 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`, 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 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 (boot after power failure), distributed at boot time across all possible `struct proc` processes, each holding the maximum possible number of file descriptor pointers (i.e., `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 ref count is equal to the number of tokens 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're 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 ref count 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.

5.7 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 call 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 in use by another process that has the file open). In this section we highlight 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 proofs represent 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 commit record 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 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 commit header. The xv6 kernel implements committing a transaction by writing a commit block to disk. If xv6 experiences a crash, xv6 will check the commit block on reboot. If there is a committed transaction in the log, it will install the blocks modified by the transaction into their home location 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 write-back cache and writes sectors asynchronously: if the disk interrupt completes, the disk guarantees that the sector is in its write-back 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) the xv6 disk driver configures the disk without `VIRTIO_BLK_F_FLUSH`, making disk operations write-through, and (2) xv6's commit log header fits in a single sector.

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 the flush feature 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 (an $\Rightarrow$ resource, §3) 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 that sector's permit 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 first sector, the proof switches the log header from the old transaction to the new one in one step; writing the second sector, whether it happens before or after the first, changes nothing that recovery reads. Together with write-through completion, this shows that when `end_op`'s write of the log header completes, the commit record 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 exceeds `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 don't 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 wasn't 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 log 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 reference 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 `namei`'s `iupdate` 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.

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 isn't 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 wasn't reused for a different block number based on the fact that the kernel doesn't 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 block 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 is 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 counts, and the inode's own invariant says that its `ip->nlink` field is equal to the number of outstanding ref counts 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 refcount. 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 plan 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 isn't strong enough, however, for concurrent system calls: another thread may have freed the inode and reused the `inum` for another inode. The naïve spec doesn't forbid it: the inode returned is allocated but isn't 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 by demonstrating that it runs during boot, when `ireclaim` 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 only exist 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 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 be a real bug with several manifestations, including a kernel panic (§8).

`ialloc`. When allocating an inode, the xv6 file system calls `ialloc` to get an inode number that's 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 couldn't 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 an agent introduced a new axiom `create_fresh_ty`. Our methodology of having an agent report its axioms caught this attempt (§7). In the end, it turned out that `ialloc` wasn't provable because `iput` had a bug (§8).

Proving `ialloc` after the `iput` fix turned out to still be difficult: the proof must make the argument that the allocated `inum` isn't 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 in the file system tree) but a file descriptor is still pointing to it; the agent had to argue that `iput` couldn't have freed such an inode yet because it still had 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.

This proof relies on the link-count and inode-type ghost state.

iput. The `iput` function puts an inode `ip` back into xv6's icache: 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 it is 1, it also 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.

5.8 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 4KB 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 in separation logic using the 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're 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 9 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.

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`.

kernel_text −∗
pc_is `syscall` −∗
sie_cap_gpr m n ... −∗
... −∗
((∀ m'. pc_is m[`ra`] −∗
sie_cap_gpr m' n ... −∗
callee_saved m = callee_saved m' −∗
... −∗
wp CpuLoop) ∧
stack_reclaim m[`sp`]) −∗
wp CpuLoop

Figure 9: 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.

5.9 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 for struct array indexing. In particular, the `procinit` function initializes the kernel stack for all `struct procs`, at kernel boot time, as shown in Figure 10. 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 10: Example of code from the `procinit` function in xv6 that generates 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, $M = 45^{-1} \bmod 2^{64} =$ `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 `p − proc` $= \frac{\delta}{360}$ as $(\delta \gg 3) \times M$. The proof of `procinit` in xv6 proves the correctness of this modular-inverse optimization.

5.10 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.

5.11 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 11 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 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 |kinds f| ≤ 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.

5.12 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's 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 12. `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 means that the current proof holds separation-logic resources for more physical memory than is possible on the machine.

5.13 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 13 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.

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 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.

```
kind ≜ Num | Str
kinds : string → list kind
    kinds ("%d" ++ r) = Num :: kinds r
    kinds ("%s" ++ r) = Str :: kinds r
    kinds ("%%" ++ r) = kinds r
    kinds (c :: r) = kinds r    (any other c)
    kinds "" = []

kind_res v Num ≜ True
kind_res v Str ≜ v = 0 ∨ ∃ s. v ↦vMemString s

args m f ≜ ⨷_{j↦k∈kinds f} kind_res m[a(j+1)] k

|kinds f| ≤ 7 →
kernel_text −∗
pc_is printk −∗
sie_cap_gpr m ... −∗
m[a0] ↦vMemString f −∗
args m f −∗
uart_sent γ bs −∗ ... −∗
(∀ m' cs. pc_is m[ra] −∗
          sie_cap_gpr m' ... −∗
          callee_saved m = callee_saved m' ∧ m'[a0] = 0 −∗
          m[a0] ↦vMemString f −∗
          args m f −∗
          uart_sent γ (bs ++ cs) −∗
          wp CpuLoop) −∗
wp CpuLoop
```

Figure 11: Simplified spec for `printk`.

The invariant also owns the user page table, user_pt *pt*: the `satp` register and the page-table pages that make up the tree rooted at *pt*, and $pa \mapsto_{\mathsf{Mem}} b$ for every byte of every physical page mapped with the user bit set, at existentially quantified 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 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

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

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.

5.14 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.

```
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_is 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_is 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 13: 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.

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's 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's 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 `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 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 two reference implementations: the QEMU rv64 whole-system emulator, which happens to be the hardware platform targeted by xv6, and the StarFive JH7110 SoC (VisionFive2) RISC-V board, manipulated via JTAG, to gain confidence that our model matches real hardware. The goal of the conformance tester 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 they are a subset of the executions allowed by the model as defined in Rocq. This is effectively a subset relation: the behaviors of the implementation must be a subset of the behaviors of the model. This allows us to argue that our theorem, which applies to every execution of the xv6 kernel on top of the trusted model, implies that every execution on top of the reference implementation is covered 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 tester, a model that gets stuck on a certain test case counts 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 concurrent execution of the 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 the model is the contents of a 4KB page of memory; the particular test case chooses what relevant data it wants to include in that 4KB page. The conformance checker must prove that every 4KB page snapshot observed in the reference implementation (QEMU or JH7110) 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 on QEMU or JH7110, 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 (4KB page snapshots) for a single test, reflecting multiple possible race outcomes. On the model side, the proof constructs an explicit schedule of steps that leads to every possible observation.

The tests also include test cases for devices, including the PLIC, the UART, and the disk, as well as the disk's DMA machinery. The RISC-V assembly instructions directly access the device MMIO registers, set up shared DMA ring buffers, etc.

7 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 check-out of the project's git repository. The agent changes files, runs `make` to check whether the proofs are correct, modifies both specs and proofs and any other files in the directory, commits, pushes, resolves merge conflicts, etc. The actual contents of the repository is 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.

As with other agent-based software development efforts, we find it's 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 agent's memory in the shared git repo, 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 (over a 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 don't 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.

7.1 Abstractions

Agents, such as Claude Code, are effective at automating proofs, but the 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 as 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 is stated with a clear separation-logic spec, and clear invariants owning separation-logic facts about shared resources, agents are effective at doing the actual proofs.

7.2 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. As another example, when we asked the agent to add the notion of a current CPUID to every WP, to allow talking about process migration across cores, the agent devolved into rewriting every specification and proof, with no end in sight. As another example, to prove `ialloc` the agent modified the xv6 code and added a new axiom to paper over a bug (that it didn't spot). As another example, 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 weren't 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 weren't sure about, we asked the agent to go ahead and implement 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.

7.3 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 second review agent 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.

7.4 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're working on, what the implementation intends to do, etc. It's 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's hard to get the agents to complete the task at hand, and it's often unclear why the agents are unable to finish it.

To address this problem, we find it's 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 new 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's already committed in the git repository. In particular, we found it's 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.

7.5 File organization discipline

We find that it's highly effective to have a strict organization structure for the files in our project's git repo. 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 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.

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.

7.6 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 inequality 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 set inequality without using `set_solver`), which improves performance of proof builds.

Occasionally, when we asked the agent to do performance optimization, it uncovered new avenues of 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 uses 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 [94].

7.7 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 less-capable agents like Opus and Sonnet are adequate at executing individual proof tasks when they don't have to design the overall proof plan.

On the other hand, for smaller-scale tasks that don't 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

Bug	Component	Consequence
A/D writeback is a blind write	Sail model	One CPU can resuscitate a page-table entry that another CPU cleared
`iput` reclaims `struct inode` instead of inode number	fs	Racing `iput` and `ialloc` orphan an on-disk inode until the next reboot
Creating files in disconnected directories	fs	`..` names a free inode, and the kernel panics because its type is not `T_DIR`
Overflowing `nlink`	fs	Too many links to one inode overflow `nlink`; unlinking it panics
Scheduler does not reset the `push_off` count	scheduler	Scheduler re-enables interrupts, and can miss a wakeup between its scan of the process table and `WFI`
Partial `writei` is not logged	fs	Partially written block is never logged, so reads change once the buffer cache recycles it
Missing icache fence when running user code	exec, trampoline	Stale icache entries from an earlier process can be executed after `exec`
`freeproc` updates `p->parent` without `wait_lock`	proc	`p->parent` is written without the lock that protects it
Inconsistent UART TX FIFO handling	uart	`printk` racing with console writes can overflow the TX FIFO
Boot code does not enable `ADUE`	boot	On hardware that follows the spec, the kernel does not handle the A/D traps it then receives

Figure 14: Bugs found during verification.

the repo, so that the agents simply needed to match the performance problem observed in some proof against the well-known optimization techniques.

7.8 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's unlikely that human proof engineers would be interested in fixing up all of the proofs after every re-compile. 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.

8 Bugs found

In the process of specifying and proving the xv6 kernel, we found ten bugs, summarized in Figure 14: one in the Sail RISC-V model that we use as our specification of the hardware, and nine in the xv6 kernel itself. The rest of this section describes each of them in turn.

Sail model bug: TLB 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.

Kernel bug: `iput` can lose an inode. The kernel had a bug in reclaiming inodes once the inode's refcount reached zero. After the refcount reached zero, the `iput` function released the lock on the `struct inode`, in order to acquire another lock to actually drop the refcount on the `struct inode`. The `iput` function then re-acquired the lock on the `struct inode` in an attempt to then mark it as free. However, the problem is that, by that time, the `struct inode`,

representing an inode cache entry, might have been reused for a different inode number. Thus, when `iput` acquired the lock, it might have now acquired the lock on a different inode number's `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 the `ip` reference to decrement `ip->ref`. Before doing the decrement, there is a short period of time during which P1 holds no locks (between releasing the `ip->lock` and acquiring the `itable.lock`). Thus, P2 creating a new file may allocate the inode (since its type is 0). P2's `iget` in `ialloc` may succeed too, since P1's `iput` hasn't acquired the `itable.lock` yet, 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 actual inode because `ip->ref` is 1. Then, P1's `iput` decrements `ip->ref` to 0, and the on-disk inode has `type` != 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 was to reclaim the inode by its inode number, 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.

Creating files in disconnected directories. If the process's cwd is inside a directory that's deleted, and the parent directory is also deleted, then `..` refers to a free inode, and trying to access `..` causes the kernel to panic because the inode type is 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.

Overflowing `nlink`. xv6 did not check for overflow on the inode's link count. As a result, creating many links to a single inode can overflow that inode's `nlink` and wrap 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.

Scheduler did not reset `push_off` count. 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.

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` returns an error back to `sys_write`, without putting the partially updated data block into the write-ahead log. This causes the file's data block to enter an unstable state: as long as it remains in the buffer cache, reads will return the partially written data, but when the buffer cache is recycled, reads will start returning the old data. We found this bug while proving that every disk buffer write is logged to the write-ahead log.

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 icache 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.

`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`.

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, we prove there are no deadlocks or other bugs, so this aspect of `printk`'s design was no longer important.

Boot code does 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.

9 Implementation

The implementation effort took 77 days and 5,051 commits. Figure 15 breaks the development down by component. Of the 1,325,913 lines of Rocq in the tree, 324,469 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 1,001,444 lines are written by agents. 405,450 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 WP rules for individual instructions, and the ghost state and invariants. The rest of the proof is specific to xv6: 62,590 lines of specifications, 527,438 lines of proof, and 2,983 lines of linking each function proof to the proofs of other functions it calls.

Outside Rocq, the development carries 9,651 lines of Python (the ELF dumper, the decode-layer and code-catalog generators, the conformance-test harness, and the reporting tools), 10,169 lines of agent-written RISC-V assembly for the conformance-test images, and 123,837 lines of Markdown design notes in 143 files.

Trusted definitions. Rocq's kernel checks the proofs, so most of the lines shown in Figure 15 are not trusted, as long as the top-level theorem 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's statement (as opposed to its proof).

The generic adequacy theorem, `riscv_power_adequacy`, 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 524 definitions spanning 4,351 lines in our proofs, plus external dependencies (the Sail model, the dump

Component	Files	Lines
Written		
Logic, ghost state, definitions	429	405,450
Specifications (`Spec*.v`)	226	62,590
Proofs (`Proof*.v`, `Wp*.v`)	396	527,438
Linking (`Link*.v`)	228	2,983
Conformance tests	17	2,837
Sail platform hooks	2	146
Generated		
Sail RISC-V model in Rocq	2	59,874
Kernel image in Rocq	5	60,218
User program images in Rocq	19	33,844
Instruction-decode layer	219	94,711
Captured QEMU and board results	273	75,822
Total Rocq		**1,325,913**

Figure 15: Size of the development, in lines of Rocq. "Generated" files are produced by the tools in `tools/` and are never edited by hand.

of the ELF kernel, the stdpp and iris libraries, etc., as discussed in §9.2). This theorem statement's definition includes some of the CSL machinery, because its statement 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, where we prove it for a specific trace property: that the file system remains consistent (across all concurrent execution, all crashes, etc.). This adequacy corollary, `xv6_fs_adequacy`, unfolds to 461 definitions spanning 3,693 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.

Invariants. The kernel's shared state is owned by Iris invariants, allocated in 25 distinct namespaces. Twelve of them belong to the file system: the log's block bytes, 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 TSO shared memory). Four are the memory-mapped devices (the UART, the PLIC, the virtio disk, and the console wire, §2); one is the kernel page table; two are 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. Two further invariants belong to the whole-system statement rather than to any one part of the kernel.

Build time. Building the whole development takes 65 minutes on a 24-core system with Intel Xeon E5-2699 2.2 GHz CPUs and 96 GBytes of memory: 2 minutes for the Sail model and the ELF dumps, 48 minutes for the Iris proofs, 12 minutes to run `Print Assumptions` on the top-level theorem to confirm the set of assumed axioms, and 4 minutes to check the conformance-test proofs.

9.1 Changes to xv6

We made almost no changes to xv6 itself, other than fixing the bugs described in §8. 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 §8). In our verified xv6 kernel, we prove the kernel will never deadlock when calling `printk`, 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).

9.2 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` statement 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 LR and SC instructions).
- The model of how the system executes, which is built on Sail's semantics, and includes a model of TSO shared memory with handling of exclusive read-modify-write operations, introduced by us, as well as models of devices (UART and 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 (§6), 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 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).

Elapsed time	77 days
Git commits	5,051
Agent sessions / working directories	407 / 13
Sub-agent runs	2,254
Human prompts	5,178
total words / median words per prompt	131,731 / 13
Agent messages / tool calls	381,356 / 388,806
Claude running time (summed over sessions)	1,729 h
wall-clock (union of sessions)	927 h
Output tokens	190.5 M
Input tokens (131.3 B cached)	133.1 B

Figure 16: Development effort, extracted from the Claude Code transcripts.

10 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?

10.1 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 16 summarizes them. Unfortunately, part of the record is lost, because Claude Code deletes transcripts older than 30 days by default. Roughly 11% of the sessions are missing, all of them from one window of about two weeks early in the development process. The rest of the record is complete, including the project's first two weeks.

Work was spread over 407 agent sessions in 13 working directories—separate check-outs of the same repository, so that several agents could work at once (§7)—and those sessions spawned a further 2,254 sub-agent runs. Across all of them we typed 5,178 prompts, totaling 131,731 words. The typical prompt is short: the median is 13 words and the mean 25.4. 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 1,540 prompts were generated by the harness rather than by us, continuing a session that had been told to keep going with `/goal`.

Against those 5,178 prompts, the agents produced 381,356 messages and made 388,806 tool calls, and Claude was actively running for 1,729 hours. Because up to 8 sessions ran concurrently, that is 927 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 190.5 million output tokens, of which 42.2 million were thinking tokens, against 133.1 billion input tokens—almost all of them (131.3 billion) served out of the prompt cache. Claude's running time includes the time it spent waiting for Rocq, which is substantial.

The cost of this development was approximately $1600 in Claude costs (for multiple Max 20x plans), plus several multi-core development VMs on AWS and GCP.

10.2 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 17 times.

Almost all of the changes are mechanical. Across the 18 commits that updated the development to a new version of xv6, 958,816 lines of generated code changed—the kernel dump, the instruction-decode layer, the user-program dumps—along with 76,044 lines of written specification and proof, a median of 2,093 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 19 of the 20 commits;

the remaining one was pruned by Claude Code. The median bump took 35 minutes of agent time and 4 prompts; the cheapest took 3 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 19 measured bumps account for 36.5 hours—166 prompts, and 2.1% of the project's agent time.

10.3 Re-proving after Sail model changes

During the development of this project, we updated the Sail RISC-V model twice (once to incorporate a fix for the PTE A/D writeback bug, and once to change how Sail extracts axioms). The PTE A/D writeback bug required significant effort to update, 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 was much more localized and required just 1.7 hours.

10.4 Model discrepancies

We generated a total of 63 test cases covering behaviors of the CPU, page tables, interrupts, shared memory, CLINT, PLIC, UART, and disk device. The JH7110 does not implement the virtio disk device, and the UART device is different from the one in QEMU and our model, so we did not run those tests on the JH7110. 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 also does not implement hardware writeback of PTE A/D bits.

Comparing the runs generated by these tests on our two reference implementations against the model surfaced 32 discrepancies, as follows.

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 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 model can execute requests out of order (and indeed our tests observed such re-ordering). 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 re-ordered, 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 CPU tests also 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, which can represent this behavior.

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. This is a known limitation that the developers of the Sail model chose to implement, based on comments in the Sail source code.

11 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 [60, 76], UCLA Secure Unix [92], PSOS [26], and KSOS [59]. These projects verified designs or partial

implementations by hand or with early tools; Kit [8] 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 [49, 50] established that full functional correctness of a microkernel is feasible, by refinement from an abstract specification down to C and, later, to the binary [78]. 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 and Verisoft XT [4, 55] pursued pervasive verification from hardware up, on a simplified processor and with the Hyper-V proofs left incomplete.

CertiKOS [35–37] verified a multicore kernel with fine-grained locking by layered refinement, with interrupts and device drivers [15] 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 [62], Serval [63], and Nickel [82] 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).

Verified security monitors and hypervisors include Komodo [30], verified in Dafny at the level of ARM assembly; SeKVM [56] and its relaxed-memory successor [87], which verify the core of KVM on Arm across cores; the Arm CCA firmware [57]; überSpark [91]; Ironclad [41]; and recent kernels written in verified Rust such as VeriSMo [95] and Atmosphere [17].

All of the above 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.

File system verification. Verified file systems have mostly been developed as standalone artifacts. FSCQ [9, 14] introduced crash Hoare logic to specify and prove crash safety of a sequential file system, and DFSCQ [16] extended it to deferred durability and `fsync` semantics; DiskSec [43] added confidentiality. Yggdrasil [81] verified a file system by push-button symbolic execution against crash-refinement specifications, and Argosy [10] showed how to compose layers with recovery. Cogent [5] and Flashix [75] verified flash file systems, and VeriBetrKV [40] a crash-safe key-value store. Concurrent verified storage came with AtomFS [96], which proves linearizability of a concurrent (but not crash-safe) file system, and with Perennial [11], GoJournal [12], and DaisyNFS [13], which extend Iris with crash reasoning to verify concurrent, crash-safe journaling and an NFS server. Ntzik et al. [66] give a separation logic for fault-tolerant resources, and SibylFS [71] a tested specification of POSIX file system behavior.

The overall file system proof plan in MachCSL follows the approach taken in Perennial [11] and PoWER [54] 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 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 write-back 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 [6] and its symbolic executor Isla [7] 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 [65]. Earlier ISA models in the same spirit include Fox and Myreen's HOL4 model of ARMv7 [31] and the ACL2 x86isa model [34]; notably, Goel verified a system-mode program with paging enabled against x86isa [33], 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 [61], Jensen et al.'s separation logic for x86 in Coq [44], and Bedrock [18] all reason about assembly-level programs, but sequentially and over simplified machines. seL4's binary verification [78] and CakeML [53, 86] 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 [24, 25], 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 [19]. 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 [74] 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.

Cerise [32, 84] is an Iris logic at the assembly level of an idealized capability machine, with logical relations for reasoning about untrusted code; later work adds MMIO devices and interrupts in a simplified form [90]. Cerise's treatment of unknown code running arbitrary instructions is analogous to MachCSL's invariant for user-mode execution (§5), although MachCSL relies on the page table and privilege mechanism of a real ISA rather than on capabilities.

Feng et al. [28, 29] used CSL-style ownership transfer at interrupt enable and disable points at the assembly level, on an idealized machine with an idealized interrupt model. At the C level, CertiUCOS [93] verifies a preemptive real-time kernel with CSL and interrupts, and CertiKOS [15] 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. Prior kernel proofs model address translation either sequentially or not at all. Verisoft verified a paging mechanism against a simple hardware MMU [1, 3], 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 [2, 21]; 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. In the seL4 line, Kolanski's mapped separation logic [51] and Syeda and Klein's logic with an explicit TLB [85] make the page table and TLB first-class in a sequential program logic. Verified hypervisors reason about page tables and TLB invalidation across cores by refinement: SeKVM [56] and its extension to Arm relaxed memory [87], Komodo [30], and the PROSPER line, which verifies MMU virtualization for Linux on ARM against an ARMv7 ISA model in HOL4 [38, 64]. Simner et al. [83] 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.

Shared memory and memory models. Formal models of the shared-memory behavior of multiprocessors are well established: x86-TSO [77] is the canonical store-buffer model, and RISC-V's architected memory model, RVWMO [72], has formal axiomatic and operational definitions, including an operational model in Coq with data-race-freedom theorems [47, 69]. RISC-V also architects TSO itself, as the Ztso extension [72], a legal strengthening of RVWMO.

Program logics for weak memory have so far been exercised only on small programs. Ridge [70] verified a spinlock with rely-guarantee reasoning over x86-TSO. iCAP-TSO [80] 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. Iris-based logics for weak memory target language-level models such as C11 [23, 89], and AxSL [39] 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 the GPFSL logic [23]. 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 [20] 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 [87] 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 [58] verifies concurrent data structures by TSO-aware refinement, and VSync [67] 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, and xv6 is: it uses flag variables in shared memory for lock-free coordination. MachCSL instead verifies xv6 in the logic against a TSO model of shared memory: well-synchronized code is verified with SC-style reasoning, and xv6's racy code against the TSO semantics itself. To our knowledge, this is the first kernel-scale proof carried out in a weak-memory program logic, and the first CSL for TSO applied beyond locks and small data structures.

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 [15] are interrupt-driven and do not access memory; Verisoft's device models [42] use programmed I/O; and the Bedrock2 lightbulb device [24] is a stream of MMIO events. Penninckx et al. [68] 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.

Acknowledgments

Thanks to Jason Gross and Tej Chajed for inspiring us to work on agent-assisted verification. Thanks to Yun-Sheng Chang and Baltasar Dinis 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, 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. http://pdos.csail.mit.edu/6.828/xv6.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] R. Jung, R. Krebbers, J. 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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 program. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210, London, United Kingdom, June 2020.

[59] 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.

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

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

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

[74] 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.

[75] 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.

[76] 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.

[77] P. Sewell, S. Sarkar, S. Owens, F. Z. 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.

[78] 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.

[79] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, Kiawah Island, SC, Dec. 1999.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

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

[86] 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.

[87] 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.

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

[89] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 691–707, Edinburgh, United Kingdom, June 2014.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

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

[95] 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.

[96] 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.