Verified IPC in the CertiKOS Microkernel

Alex Lew

Computer proof assistants like Coq have made it possible in recent years to formally verify the correctness and security of software. Unlike software testing, in which code is run on a limited number of inputs and checked for the expected behavior, software verification allows us to prove that a piece of code will run as expected in every case. In this report, I describe efforts to contribute to a verified operating system kernel called CertiKOS, under development by Professor Zhong Shao and his lab. I worked closely with Hengchu Zhang, another undergraduate, to improve the functionality and performance of inter-process communication protocols in CertiKOS, then prove our new code correct.

In this report, I describe the tools and methods used in the project, as well as the result of my work. I then describe possible avenues of future work; as they are relatively manageable, I intend to pursue them in the next couple weeks.

1 Introduction

Formal verification of software

Most software has bugs. Generally speaking, this does not keep users awake at night: an occasional crash can be annoying, but rarely causes too much trouble. For some applications, though, it is of critical importance that software be correct. In domains as varied as health,
space travel, and the military, the difference between buggy and bug-free can be a matter of life and death.

In industry, quality assurance (finding and quashing bugs) is typically performed using automated and manual testing. Before code can be shipped, it must produce expected results on a variety of inputs. When code changes, engineers check that it still passes all the tests. While testing can be effective, it is not fool-proof. Engineers who write the tests must imagine every possible use case. Furthermore, in (for example) open-source software, a malicious actor could write faulty code that nonetheless passed every test, by hard-coding expected behavior on the inputs tested.

Formal software verification techniques eliminate most of these problems (and, it should be said, create others – maintaining formal proofs of a large code base can be unwieldy). At a high level, formal verification seeks to prove, in a mathematical sense, that certain properties are true of a piece of a code. Whereas in testing, a programmer might write several test cases to check that, for example, a calculator performs integer addition correctly (checking for edge cases such as adding negative numbers, adding zero, and adding two very large numbers), in software verification, the programmer would prove the theorem that for all integers \( a \) and \( b \), the sum function on the calculator outputs \( a + b \). If the theorem is in fact not true (because of, for example, integer overflow), the theorem will be unprovable, and the programmer will either have to modify the calculator’s code or change the theorem: for all integers \( a \) and \( b \), if \( a + b < \text{INT\_MAX} \), the sum function outputs \( a + b \). In testing, it is possible to miss error cases like this; in software verification, you must formally state any exceptions to the rule in order to prove the rule correct.
A verified kernel

Guarantees on software like desktop calculators are somewhat useless unless the entire stack on which the software is built can be trusted. We must trust that the compiler produces a correct translation of, say, C, to assembly and then machine code; we must trust that the operating system is correct and secure; and that the hardware too is flawless. Professor Zhong Shao’s lab is devoted to tackling the lowest level on the software stack: the operating system. CertiKOS is a verified operating system kernel. A focus of the project is modularity and scalability: other attempts to verify an operating system have been arduous and long, due to the complexity of verifying something with so many interdependent parts. CertiKOS uses new proof techniques that make both the OS code and the proofs modular and easily scalable.

My project was to update one piece of the kernel, implementing and verifying new and improved inter-process communication functionality. The rest of this paper describes the tools and techniques used to do so.

2 Coq: Types and Proofs

One tool that has been essential in verifying the CertiKOS kernel is Coq, a computer proof assistant based on the programming language Gallina. Proof assistants are tools that help formalize and prove lemmas and theorems. Most of the common proof assistants today, including Coq, rely on an insight from Howard and Curry in the late sixties – an insight today referred to as the Curry-Howard Correspondence.

The insight relates logic and computation as follows: programs can be understood as proofs, and types of programs as propositions. In computation, a type can be understood as a set of programs (or functions); similarly, a proposition can be understood as a set of proofs for that proposition (where false propositions are empty sets). Then familiar operations on sets (such as
unions and cross products) take on new meanings. A tagged union type (whose values are either of type \( A \) or type \( B \)) corresponds to an ”or” of two propositions; if there is a proof (program) for either \( A \) or \( B \), then there is a proof (program) for ”\( A \) or \( B \)” (the union of \( A \) and \( B \)). Similarly, a program of a tuple type \((A, B)\) produces both a program or value of type \( A \) and a program or value of type \( B \). This corresponds to a logical ”and” – a proof for \( A \) and a proof for \( B \) constitute a proof for the proposition ”\( A \) and \( B \)”.

Similarly, function types correspond to logical implication. A function of type \( A \rightarrow B \) takes in a value (proof) of type \( A \) and produces a value (proof) of type \( B \); the function itself, then, is a proof of the logical proposition \( A \rightarrow B \). The logical negation of a proposition \( A \), \( \neg A \), can be represented as a function from \( A \) to the False type, \( A \rightarrow \text{False} \), where False is an empty type.

Proof assistants like Coq make use of this correspondence by making available to users a richly typed functional programming language like Gallina, and providing semantic tools to help users interpret some of the types and functions in their code as propositions and proofs. To see this in action, let’s consider a basic definition of the natural numbers in Coq, as well as a statement of a lemma about them.

First, we define a type for natural numbers:

\[
\text{Inductive nat : Set :=}
\]
\[
| \text{O : nat} \\
| \text{S : nat \rightarrow nat.}
\]

This is an inductive definition. We define a new inductive type (”Set”) called ”\text{nat}”, and then enumerate the elements: every element is either of the form ”\text{O}” (zero), or ”\text{S } n” for some other natural number \( n \) (the notation \( S : \text{nat \rightarrow nat} \) means that \( S \) is a function taking natural numbers to new natural numbers; therefore if \( n \) is of type nat, so is \( Sn \). \( S \) here stands for ”successor”. So the number one would be represented as \( S \text{O} \), two as \( S (S \text{O}) \), etc.

We can use a very similar definition to define the set of propositions ”\( n \) is even.”
Inductive even : nat → Prop :=
   | even_O : even 0
   | even_S : forall n, even n → even (S (S n))

Again, we define a type by showing all of the ways to construct a value of that type. Here we are in fact defining a set of types (a set of propositions): "even n" is a different proposition for each n. To create a proof of "even n", you may use either the constructor even_O, which, being of type even O, can only prove that zero is even, or even_S, which can be used to create a proof of type even (S (S n)) given a proof of even n.

We can then write things like this:

Lemma two_is_even : even (S (S 0)).
Proof.
  apply (even_S (O) (even_O)).
Qed.

The term "even_S (O) (even_O)" is called a "proof object"; it is a value of type "even S (S O)" and its existence proves that two is even. In practice, we spend little time hand-coding proof objects in Coq; there is a rich library of "tactics" – essentially macros – that help construct proofs for us.

Here, we defined natural numbers; in software verification, we use similar inductive definitions to define C expressions, statements, and programs. We can then state propositions and prove lemmas about the programs or functions implemented by various blocks of C code.

3 Abstraction layers

CertiKOS is not the first attempt to formally verify an operating system kernel. What is groundbreaking about CertiKOS is the scalability of the proof techniques the team is using. Where other verified kernels have generally been small (microkernels) and taken decades to complete, CertiKOS is more ambitious, and its development has been fast. The breakthrough is to model
formally, in the proofs, the process most programmers use to write complex and scalable software: multiple layers of abstraction.

In CertiKOS, each layer consists of a number of functions that call only functions from lower layers. Furthermore, when proving lemmas and theorems about the functions in a given layer, only the specifications of and lemmas proved about the functions in lower layers are available to you. There are currently dozens of layers in the CertiKOS kernel. Doing things this way makes the proof techniques much more scalable.

4 Verifying IPC

For my project, I took on an essential piece of any operating system: the protocol for interprocess communication. Working closely with Hengchu Zhang, another undergraduate, I created and iterated on a working C prototype of an IPC algorithm, wrote specifications in Coq for each new C function, proved that each C function indeed implemented its specification, and finally proved several theorems about the specifications, e.g. that they all preserved certain invariants.

IPC in CertiKOS is exposed to user processes as a pair of function calls: one for sending, and one for receiving. Sending must occur before receiving. The sender specifies the process ID (an unsigned integer less than 64) of the receiver, a number of words to send, and a buffer where the data to be sent is stored. In the current version, the buffer passed in must be aligned at a page boundary, though this is something we hope to change in the next couple weeks. (One can obtain such a buffer using GCC compiler directives.) When the send function is called by process \( i \), the kernel puts the process to sleep and updates the \( i \)th entry of a 64-entry array called the "synchronous channel pool." Each entry contains the physical address of the buffer being sent, the number of items to be sent, and the process to which the message is being sent. When the receiver call the receive function and asks for a message from a certain process \( i \), the kernel
checks the $i$th entry of the pool to ensure that process $i$ is indeed trying to send a message to the receiver. The receiver passes in a virtual address to a buffer as well as a number of words it would like to receive. The kernel copies the message into the buffer, then wakes the sender.

For each C function, we wrote a spec in Gallina, the language that backs Coq.

Here’s a relatively involved example, for the receive function:

```gallina
Function syncreceive_chan_spec(fromid vaddr rcount : Z)(adt : RData) : option (RData * Z) :=
match (pg adt, ikernt adt, ihost adt, ipt adt) with
| (true, true, true, true) =>
match ZMap.get fromid (abtcb adt) with
| AbTCBValid st _ =>
if ThreadState_dec st DEAD then
Some (adt, 1024+2)
else
match ZMap.get fromid (syncchpool adt) with
| SyncChanValid to spaddr scount =>
if zeq (Int.signed to)(cid adt) then
let arecvcount := Z.min (Int.signed scount) rcount in
match get_kernel_pa_spec (cid adt) vaddr adt with
| Some rbuffpa =>
match flatmem_copy_spec arecvcount (Int.signed spaddr) rbuffpa adt with
| Some adt1 =>
let adt2 := adt1
{syncchpool := ZMap.set fromid (SyncChanValid (Int.repr num_chan) Int.zero
(Int.repr arecvcount)) (syncchpool adt1)} in
match thread_wakeup_spec fromid adt2 with
| _ => None
end
| _ => None
end
| _ => None
end
else
Some (adt, 1024+3)
| _ => None
end
| _ => None
end
| _ => None
end.
```
The spec is essentially a rewriting of the original function, with a few important differences. First, it takes in an additional argument, beyond the three arguments of the C "receive" function (fromid, vaddr, and rcount): adt, of custom type RData, which is a representation of the state of the machine. A (possibly modified) version of the state is returned by the function, in addition to the integer return value also present in the C code. Second, when the behavior of the C code is undefined (i.e., we as verifiers do not care what the behavior will be), the spec returns None.

Note that other specs are called from within this spec. These functions are all defined and verified at a lower abstraction layer, so they are guaranteed to have been proven correct before we have to prove anything about this new code.

For each spec, we prove a series of lemmas. For example, for we prove for syncreceive_chan_spec that:

**Lemma syncreceive_chan_exist:**
\[\forall s f \text{habd habd'} \text{labd fromid vaddr count i},\]
\[\text{syncreceive_chan_spec fromid vaddr count habd} = \text{Some (habd', i)} \rightarrow \text{relate_AbData s f habd labd} \rightarrow \exists \text{labd', syncreceive_chan_spec fromid vaddr count labd} = \text{Some (labd', i)} \wedge \text{relate_AbData s f habd' labd'}\]

This lemma states that if a high-level representation of the machine state is related to a low-level representation, executing syncreceive_chan_spec on both will maintain that relationship.

Similarly, we prove a lemma that each spec preserves a number of important invariants.

The most involved proofs come when proving that the C code for a function actually implements a spec. For syncreteceive_chan, we state the lemma like this:

**Lemma syncreceive_chan_body_correct:**
\[\forall m \text{d' env le fromid vaddrval rcountval val},\]
\[\text{env} = \text{PTree.empty} \rightarrow \]
\[\text{PTree.get pid le} = \text{Some (Vint fromid)} \rightarrow \]
\[\text{PTree.get vaddr le} = \text{Some (Vint vaddrval)} \rightarrow \]
\[\text{PTree.get rcount le} = \text{Some (Vint rcountval)} \rightarrow \]
\[\text{high_level_invariant d} \rightarrow \]
\[\text{syncreceive_chan_spec (Int.unsigned fromid)(Int.unsigned vaddrval)}\]
\[\text{(Int.unsigned rcountval)d} = \text{Some (d', Int.unsigned val)} \rightarrow \]
\[0 \leq \text{Int.unsigned fromid} < \text{num_proc} \rightarrow \]
exists le',
exec_stmt ge env le ((m, d); mem) syncreceive_chan_body E0 le'(m, d') (Out_return (Some (Vint val, tint))).

This states that if syncreceive_chan_body (the C code) is executed while the machine is in state \( d \), with memory \( m \), local environment \( le \), and empty global environment \( env \), with arguments fromid, vaddrval, and rcountval, and if the spec (called on those arguments) returns the value \( val \) along with a modified state \( d' \), and the fromid passed in is between 0 and 64 (num_proc), then the return value from the C code will be \( val \) and we will be in the new state \( d' \).

The proofs themselves can be inspected in the source files submitted for this project.

5 Future Work

In the next couple weeks, I plan to remove some of the restrictions present in the current IPC algorithm. For example, I will add code that will obviate the requirement that the sender’s buffer be aligned with the page boundary, as well as the requirement that the transfer size be less than 1024 words.

Depending on how difficult this is, I may have time left over to implement some performance enhancements. Microkernels like SeL4 have stringent IPC performance requirements, due to the fact that almost all services are implemented outside the kernel. (For example, drivers are user-space processes in an seL4-based operating system, so any communication with devices must go through the IPC pathway.) Therefore, there are many ideas embedded within the seL4 code that may be worth implementing in CertiKOS. Using kernel-space memory for IPC buffers is one example – this avoids walking the page table to convert between address spaces for different processes. Another example is passing smaller messages in registers instead of buffers.