CS490 Project Proposal

Alex Lew

I will be working with Professor Zhong Shao and another student, Hengchu Zhang, to implement a fully verified Inter-Process Communication module in the CertiKOS microkernel. During the first phase of the project, Hengchu and I will work together to implement and prove correct a simple and fast IPC algorithm. Afterward, we will work individually on separate enhancements to the algorithm.

I offer here an overview of computer-assisted formal proof verification techniques and tools, the CertiKOS microkernel and the motivation for it, and the specific area I will be working on (inter-process communication).

1. Coq Formal Proof Verification

“Computer-assisted proof verification” is exactly what it sounds like: the use of a computer to verify that a rigorous mathematical proof is indeed correct. Computer scientists have discovered a remarkably elegant way of achieving this, based upon a series of observations by logician Haskell Curry and proof theorist William Howard. The so-called Curry-Howard Correspondence is a relation between logical propositions (e.g., “not all odd numbers are prime”) and programming-language types. A proposition is true if and only if the corresponding type is inhabited; that is, if and only if there exists some program code (a “proof object”) that computes a value of the desired type.
This sounds very abstract but in fact is quite simple. To get a handle on it, consider type constructors that are common in functional programming languages. They all have clear analogues in formal logic. For example, given two types $a$ and $b$, we can create a third type, $a \rightarrow b$, a function from $a$ to $b$. That third type represents the proposition $a$ implies $b$: a value of type $a \rightarrow b$ is an algorithm for turning a value of type $a$ (a proof of proposition $a$) into a value of type $b$ (a proof of proposition $b$). If such an algorithm exists, then $a$ implies $b$. Similarly, given two types $a$ and $b$, we can create a third type $(a, b)$, a tuple containing an element of type $a$ and an element of type $b$. Clearly this corresponds to the proposition $a$ and $b$: produce a proof for $a$ and a proof for $b$ to get a proof for $a$ and $b$.

Coq is a programming language with an incredibly rich type system, streamlined for taking advantage of the Curry-Howard Correspondence. It provides syntactic sugar for declaring types that correspond with propositions. Then you can use a simple type-checking algorithm, the correctness of which is easily verified, to verify that a proof of some proposition (a value of some type) is correct (actually type-checks to the desired type). Coq also provides a DSL called LTAC with which one can develop “proof tactics,” which can help automate the process of creating proof objects.

In CS428, Professor Shao explained how one might go about using Coq formalizing the semantics of imperative programming languages like C. (C’s full semantics don’t actually stand up to rigorous formalization, so researchers tend to use a subset of C
that behaves deterministically and is nicely formalized.) Once the requisite definitions have been stated, one can use Coq to verify and prove theorems about pieces of code.

2. CertiKOS “microkernel”

Using techniques described in the previous section, we can prove that the code behind, say, an operating system kernel is completely bug-free. The applications of this are to some degree self-evident. The kernel is the software that operates at the lowest level of abstraction, directly on hardware. A bug in the kernel can lead to unexpected behavior in any user-space application running on the kernel.

The idea of verifying a microkernel is not new – in fact, a verified microkernel called seL4 has been in development for many years now. But CertiKOS takes a different, more scalable approach. The architecture of CertiKOS is based on the observation that programmers already think in terms of layers of abstraction. We use software we didn’t write, and rely on documentation (in English or other natural languages) to tell us what that software does. Those manuals define an interface and claim that a certain piece of software implements that interface correctly. But they don’t guarantee that the software actually works as intended. There could still be bugs.

CertiKOS uses “deep specification” and formal abstraction layers to capture a rigorous version of the informal idea of the abstraction layer. When trying to prove things about a piece of code, it is first broken up into abstraction layers, where each
layer hides certain implementation details and exposes only the interface that upper layers will need. However, unlike interfaces in most programming languages, which tell you only superficial information about the implementation (e.g., type information), these interfaces are actually deep specifications, defining *everything* a user of the abstraction layer might want to know. Coq is then used to prove that the implementation actually does implement this specification.

The benefit of this is that code becomes much more modular, and at each layer, we need only prove that everything works as intended *given that* the previous layer is correctly verified. This has allowed for much faster progress on CertiKOS than on seL4.

### 3. Inter-Process Communication

In a microkernel like CertiKOS, inter-process communication (IPC) is very important: since almost everything runs in user-space, communication that would normally happen within the kernel must instead use IPC. It is essential, then, that any solution be fast.

The goal of this project is to find a fast algorithm for IPC that is also elegant and simple enough that it can be slotted in well to the multi-layered architecture of CertiKOS, and proven in Coq without too much trouble (though I anticipate the proofs will take quite a while even with the best of algorithms). As of right now, the
plan is to implement synchronous IPC, in which if one process sends information, it cannot continue execution until the recipient process has received the information.

4. Deliverables

I plan to deliver a CertiKOS module for IPC with a complete specification / verification and a write-up. I also hope to have begun to explore enhancements to that IPC algorithm, and to have at least a prototype of some fancier IPC system, and the beginnings of a proof, ready by the end of the term.