Are You Sure?
Extending a certified operating system

John Whittaker, advised by Zhong Shao
April 30, 2014

Abstract
This is the final report for a project completed by the author in partial fulfillment of the Computer Science (B.S.) major at Yale University, done under the guidance of Prof. Zhong Shao of the Department of Computer Science. The author has extended a small portion of his advisor’s research work on a groundbreaking operating system called CertiKOS pertaining to inter-process communication (IPC). CertiKOS is a certified operating system, meaning that it uses mathematically-based proof tools, like the computer proof assistant Coq, to formalize and prove statements about the properties of the operating system and of application programs that are running on it. In this report, the author introduces the reader to Coq, CompCert (a certified compiler), and CertiKOS and explains the work he has done on modifying the original source code of the IPC implementation and the changes he has had to make to the Coq proofs pertaining to IPC as a result of modifying the source code. The author presents informal proofs of properties about the new IPC implementation, such as the termination of functions defined and the memory state resulting from calling functions defined in the relevant files. The informal proofs presented follow the formalized proofs but are much shorter and designed to be easily read. The reader is encouraged to view the source code and Coq proofs, which can be found on the web page for this project. Topics covered include machine-aided proof-writing, reasoning about the execution of program code, reasoning about loop in programs, and the utility of formally verified software. It is the author’s hope that this report will be useful to others who might be interested in Coq, CompCert, or CertiKOS.

1 Introduction
Our increasingly technology-driven world, in which program bugs can cost millions of dollars or even people’s lives, has led to the rise of formal methods in software development. Even guaranteeing the correctness of the source code of an application program may itself not be enough, as a bug in the infrastructure on which a program relies, including compilers and operating systems, may cause a perfectly well-written program to behave abnormally. Such formally verified systems are not merely an academic fancy but are increasingly becoming more and more necessary, especially with the proliferation of software into so-called
“life-critical” and “mission-critical” areas, including military and medical applications [1]. In fact, the ease with which a malicious hacker can gain control over a car’s acceleration or breaking systems using flaws in the Bluetooth or other communication systems—which presumably should themselves have no access to the critical systems of an automobile—is astonishing. Although the certified compiler CompCert has been largely relegated to academia, there has been an increasing interest from industry, including the airplane manufacturer Airbus [2], in the extra layer of reliability that CompCert provides.

Prof. Zhong Shao and his team of researchers at Yale University are in the process of developing a certified operating system [10]. For my senior project, I have extended a small portion of that operating system, namely the inter-process communication implementation.

In addition to describing the work I have done on my senior project, it is my intention in writing this paper that it may be understandable to anyone without any special knowledge and that it could potentially serve as an introduction to anyone interested in using the Coq proof management system or to future students who might be interested in contributing to Prof. Shao’s research.

2 Coq

Coq is a “formal proof management system” [3]. One can use Coq to perform certain tasks formerly confined to pencil and paper or left to human intuition, including, but not limited to, the formalization and proving of mathematical statements, the formalization of the semantics of a programming language, and the formalization of reasoning about a program and adherence of a program to a specification. Indeed, Coq provides a level of rigor beyond that of more informal mathematical proofs, and for its “influential role in formal methods, programming languages, program verification and formal mathematics,” Coq has been named recipient of the 2013 Software System Award [5]. Coq has been used to prove the Four Color Theorem (stating that a planar map can be colored using four colors such that no two adjacent regions share the same color) [4], and it is used in the verification of the CompCert compiler and the CertiKOS kernel. Despite Coq’s use in formalizing and proving complex theorems about mathematics or computer programs, the basics are rather simple to pick up, illustrated by the following proof.

Theorem example : forall (a b:Z), (a + b) - (b + a) = 0.
Proof.
intros a b.
(* Theorem Zplus_comm : forall n m:Z, n + m = m + n. *)
rewrite Zplus_comm with (n:=b) (m:=a).
(* Lemma Zminus_diag : forall n:Z, n - n = Z0. *)
apply Zminus_diag with (n:=a+b).
Qed.

The above is a proof that for all a and b, where a and b are integers (called Z in Coq), (a+b) – (b+a) = 0. The first line of the proof introduces universal variables a and b (which correspond to the a and b of the statement of the theorem but could also be named anything else). The next line is a comment, showing
a (pre-proven) theorem in Coq’s standard library that integer addition is com-
mutable. The line after uses the theorem Zplus_comm to rewrite (b+a) with
(a+b). Now that we are trying to prove that (a+b) - (a+b) = 0, we can use
another theorem in Coq’s standard library that states that any integer n minus
itself is equal to 0 (written in the theorem as Z0). We apply Zminus_diag with
n as a+b, which ends the proof. Instead of having to keep track of the progress
of the proof in one’s head, interactive tools such as the ProofGeneral extension
to Emacs will automatically keep track of and display one’s progress. So, for ex-
ample, after the line rewrite Zplus_comm with (n:=b) (m:=a), ProofGeneral
shows the following status:

```
a : Z
b : Z

============================
a + b - (a + b) = 0
```

This means that there are two variables a and b, both of type Z (integer), and the
statement below the double line is the current goal we are trying to prove. For
such a simple proof it is not necessary, but for longer, more complicated proofs,
writing Coq proofs in ProofGeneral’s interactive mode is extremely helpful.

Although a bug within Coq itself could cause Coq to accept invalid proofs,
Hales notes in [9] that the error rates of Coq and similar proof management
systems are “orders of magnitude lower than error rates in most prestigious
mathematical journals.” So we can use Coq to provide a further layer of certainty
beyond that which traditional proofs provide.

## 3 CompCert

CompCert is a verified C compiler, meaning that it has been proven, using Coq,
that “the generated machine code behaves as prescribed by the semantics of
the source program” [6]. Compilers, like other programs, contain bugs, and
just as with many applications, we accept a certain degree of bugs in exchange
for efficiency or affordability. However, for certain “mission-critical” applica-
tions, including medical and military applications, bugs may be unacceptable,
and even if one uses formal methods to verify the source code of an applica-
tion program, that effort is meaningless if the compiler used transforms correct
source code into incorrect machine code. And even for applications that are
not “mission-critical,” compiler bugs can be very subtle and difficult to track
down. This is where CompCert comes in. Although the algorithms it uses to
compile code are not revolutionary, its guarantee to preserve the semantics of
a source program written in a subset of C called Clight, described in the next
section, whenever it successfully compiles a program without emitting an error
message, is revolutionary. Not only is CompCert correct, it is also fairly effi-
cient, with compile times within a factor of 2 of gcc -01 and compiled code
only 7% slower, on average, than gcc -01 [6]. Therefore, in addition to being
of considerable academic interest, CompCert could prove itself to be valuable
in industry, in cases where the reliability provided is more important than the
fairly small difference in efficiency between CompCert and gcc.
4 Clight

Clight is a subset of the C language, which includes most of the important features of C, including pointer arithmetic, structs, unions, conditional statements, loops, switch statements containing a default case, break, continue, and return \[7\]. Some of the omitted features include goto statements, typedef definitions, string literals, which are changed into global initialized character arrays, and the type qualifiers const, volatile, and restrict, which are erased during parsing \[7\]. The semantics of Clight have been formalized in Coq, allowing one to use Coq to state and prove properties about Clight programs \[7\].

\[ s ::= \text{skip} \quad \text{(empty statement)} \\
| a_1 = a_2 \quad \text{(assignment)} \\
| a_1 = a_2(a^*) \quad \text{(function call)} \\
| a(a^*) \quad \text{(procedure call)} \\
| s_1; s_2 \quad \text{(sequence)} \\
| \text{if}(a) \ s_1 \ \text{else} \ s_2 \quad \text{(conditional)} \\
| \text{while}(a) \ s \quad \text{("while" loop)} \]

Above is an (abbreviated) abstract syntax for Clight expressions from \[7\], and below is its formalization in Coq from \[8\], also abbreviated.

\[
\text{Inductive } \text{statement} : \text{Type} := \\
| S\text{skip} : \text{statement} \\
| S\text{assign} : \text{expr} -> \text{expr} -> \text{statement} \\
| S\text{set} : \text{ident} -> \text{expr} -> \text{statement} \\
| S\text{call} : \text{option ident} -> \text{expr} -> \text{list expr} -> \text{statement} \\
| S\text{sequence} : \text{statement} -> \text{statement} -> \text{statement} \\
| S\text{ifthenelse} : \text{expr} -> \text{statement} -> \text{statement} -> \text{statement} \\
| S\text{loop} : \text{expr} -> \text{statement} -> \text{statement} -> \text{statement} \]

Using CompCert’s clightgen tool, one can parse C programs into Coq-readable code that obeys the Clight abstract syntax. For example, consider the following simple C function.

\[
\text{int zero(int x) \{} \\
\quad \text{if} (x) \{} \\
\quad \quad x = 0; \\
\quad \} \\
\quad \text{return } x; \\
\}
\]

Using clightgen on the above function will result in the following as the body of the function (along with some function and program metadata that has been omitted).

\[
(* \text{Definition of the body of the above function. } *) \\
\text{Definition zero_body := } (S\text{sequence} \\
\quad (S\text{ifthenelse} (E\text{tempvar } tx \ tint)) \\
\quad (S\text{call} (E\text{tempvar } tx \ tint) (E\text{tempvar } tx \ tint)) \\
\quad (S\text{assign} (E\text{tempvar } tx) (E\text{constant 0})) \\
\quad (S\text{set} (E\text{ident } \_x) (E\text{constant 0})) \\
\quad (S\text{ifthenelse} (E\text{tempvar } tx \ tint) (S\text{assign} (E\text{ident } \_x) (E\text{constant 0})) (S\text{set} (E\text{ident } \_x) (E\text{constant 0}))) \\
\quad (S\text{sequence} (S\text{ifthenelse} (E\text{tempvar } tx \ tint) (S\text{assign} (E\text{ident } \_x) (E\text{constant 0})) (S\text{set} (E\text{ident } \_x) (E\text{constant 0}))) (S\text{call} (E\text{tempvar } tx \ tint) (E\text{tempvar } tx \ tint)) \\
\quad (S\text{assign} (E\text{ident } \_x) (E\text{constant 0})) \\
\quad (S\text{set} (E\text{ident } \_x) (E\text{constant 0}))) \\
\quad (S\text{ifthenelse} (E\text{tempvar } tx \ tint) (S\text{assign} (E\text{ident } \_x) (E\text{constant 0})) (S\text{set} (E\text{ident } \_x) (E\text{constant 0}))) \\
\quad (S\text{sequence} (S\text{ifthenelse} (E\text{tempvar } tx \ tint) (S\text{assign} (E\text{ident } \_x) (E\text{constant 0})) (S\text{set} (E\text{ident } \_x) (E\text{constant 0}))) (S\text{call} (E\text{tempvar } tx \ tint) (E\text{tempvar } tx \ tint)) \\
\quad (S\text{assign} (E\text{ident } \_x) (E\text{constant 0})) \\
\quad (S\text{set} (E\text{ident } \_x) (E\text{constant 0}))) \\
\text{)} \\
\]

4
The above Coq-format Clight is now amenable to proving various properties about, including, but not limited to, safety, termination, whether or not it modifies memory, and the outcome of executing the function body, in this case returning a value, which we can prove to be the integer 0. Notice that the original if statement does not contain a matching else statement, as required by Clight. In order to remedy that, the parser makes the else branch a skip statement. In the Appendix, in the file report_examples.v, I have included a sample proof involving the above function body, showing the result of executing the statement body is a return of the integer 0. In short, this can be shown by case analysis on $x = 0$. In the case where $x = 0$ is false, the if condition is true, $x$ is assigned to 0, and the value of $x$ is returned. In the case where $x = 0$ is true, the body of the else statement, in this case skip, is executed followed by returning the value of $x$, which we can prove to be 0 since the condition $x = 0$ held true at the point of the if, and there have been no assignments to $x$ in the interim. For the full Coq proof, please see report_examples.v in the Appendix.

5 CertiKOS

CertiKOS is a certified operating system under development by Prof. Zhong Shao and his team at Yale University. The kernel is compact and “composed of modular, replaceable, and individually certifiable plug-ins” [10]. For the sake of modularity, extensibility, and organization amenable to proof writing, the CertiKOS kernel is split into 37 abstraction layers, which have limited dependencies on each other [10]. Prof. Shao and his team use Coq to formalize the kernel code and to state and prove properties about the various layers and their interactions with each other. Beyond being a tremendous academic undertaking, the certification of an operating system that is safe, secure, and reliable would be groundbreaking, in fields as varied as automobile embedded systems and medical applications.

6 Project Overview

For my senior project, Prof. Shao has generously given me access to his research, source code, and Coq proofs. I have extended the inter-process communication capabilities of the original system. Inter-process communication refers to methods by which multiple processes can communicate with each other [11]. There are many different ways to implement inter-process communication, and even within a single operating system there could be multiple ways of passing information to other processes. Within the original CertiKOS implementation, inter-process communication is achieved by using system calls to send or receive a single integer of data. Although there is certainly nothing wrong with passing one integer at a time to communicate with other processes, using an entire buffer would add a level of convenience for application programs. Of
course, before beginning extending the CertiKOS kernel, I first had to familiar-
ize myself with the kernel source code, Coq proofs about the source code, and
auxiliary Coq files, including those pertaining to reasoning about while loops,
whether they terminate, and their effects when they do terminate. For those in-
terested in seeing some of the practice proofs I wrote about simple C programs,
please see my code base on the Computer Science Special Projects web page:

7 Original IPC Implementation
The original IPC implementation consisted of calling functions declared in ipc.c.
These functions include ones for checking if a channel (i.e. process) is ready
to receive data, for sending data, for receiving data, and for initializing the
channel pool. The functions defined in ipc.c make up the “front-end” of the
inter-process communication implementation, and those functions call functions
defined in ipc_intro.c, which make up the “back-end” of the inter-process com-
munication implementation. It is the back-end functions that actually read and
write from memory, and the front-end functions ensure that the back-end func-
tions are used properly, i.e. that indices are within correct bounds and that
there are not conflicts between processes trying to send data to the same pro-
cess at the same time. The original ipc.c and ipc_intro.c are attached to the
end of this report in the Appendix. However, some excerpts follow.

7.1 Back-End
The back-end of the IPC implementation, as defined in ipc_intro.c, is very
simple, using a struct for each channel (i.e. process), containing an isbusy
field, specifying whether a channel is ready to receive data (when isbusy is
false) or whether it cannot currently receive data (when isbusy is true), which
is the case when someone has sent data to a process but the process has not
yet read that data. The following is the struct, followed by an array, where
NUM_CHAN is the maximum number of processes that could exist.

```c
struct ChanStruct {
    unsigned int isbusy;
    unsigned int content
};
struct ChanStruct CHPOOL_LOC[NUM_CHAN];
```

The functions in ipc_intro.c are likewise very simple. The following function
is used to set the content field of a specific ChanStruct.

```c
void set_chan_content(unsigned int chid, unsigned int content) {
    CHPOOL_LOC[chid].content = content;
}
```

The above function assigns the parameter content to the content field of the
chidth ChanStruct in the CHPOOL_LOC array. As you can see, the function does
not check to make sure that the value of chid is a legal valuable, i.e. less than
NUM_CHAN (which happens to be 64): that is the responsibility of the calling function (in ipc.c). The function set_chan_content, like the other functions in ipc_intro.c is simply a straightforward way to indirectly modify the memory used to share data between processes. Application programs are not allowed to call functions defined in ipc_intro.c, as they could, for example, specify an illegal chid argument; instead, they must call the functions in ipc.c, which (described below) will enforce restraints, such as having a legal chid value or making sure not to write to a channel that is currently busy. Of course, the functions in ipc_intro.c and ipc.c could be combined, but separating out the functions that directly read and modify memory (ipc_intro.c), from those that are called by application programs and verify that the various restraints are satisfied (ipc.c) does not only make the code clean and logical, but it makes it easier to modify (which was my job!) and easier to reason about and write proofs (also my job!). The other functions in ipc_intro.c are very similar and can be seen in the Appendix of this report.

7.2 Front-End

The front-end functions (defined in ipc.c) are those that are called by application programs, so of course no assumptions can be made about whether parameters, such as the id of the process to which we are sending data is valid or not. Consider the following function for sending data to a channel.

```c
unsigned int send(unsigned int chid, unsigned int content) {
    unsigned int info;
    if (0 <= chid && chid < NUM_CHAN) {
        info = get_chan_info(chid);
        if (info == 0) {
            set_chan_info(chid, 1);
            set_chan_content(chid, content);
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}
```

The above function first verifies that the chid is a legal value (i.e. less than NUM_CHAN, and if it is not, returns 0 to indicate failure. Then it gets the target channel’s info (which in ipc_intro.c is represented by the isbusy field of a ChanStruct). If the info is 0, then the function proceeds, and otherwise (when the channel is currently busy), it returns 0 for failure. In the case where info is zero, we call set_chan_info to set the channel’s isbusy field to 1, call set_chan_content to set the content to the desired value, and then return 1 for success. Note that because this is kernel code, it will not be possible for two processes to be inside the send function at the same time, so we can think of the entire function as being atomic. Again, the other functions in ipc.c are fairly similar and can be seen in the Appendix.
8 New Implementation

In the new IPC implementation, I have replaced the single integer used to store `content` with an array of integers. Reading and writing still occur one integer at a time, but the reads and writes are now to one of the integers in the `content` array instead of to a solitary `content` integer. Of course, one could directly use pointers to copy arrays of integers (like, for example, the C function `memcpy`). However, this makes it more difficult to prove safety properties, and indeed the recent Heartbleed Bug that has been appearing in the news [13] takes advantage of a software flaw in OpenSSL allowing access to read memory beyond that which was anticipated, as illustrated succinctly in [14]. The new implementation pairs the convenience of using an array for inter-process communication with a simplicity that allows for more straightforward proofs of its validity and helps avoid some of the nasty issues, including those like the aforementioned Heartbleed Bug, that dealing with pointers can cause.

8.1 Back-End

The back-end (`ipc_intro.c`) is changed so that instead of having a single `unsigned int` to transmit data from one process to another, there is now an entire array of size `IPCBUFSIZE`.

```c
struct ChanStruct {
    unsigned int isbusy;
    unsigned int content[IPCBUFSIZE];
};

struct ChanStruct CHPOOL_LOC[NUM_CHAN];
```

Like before, there is a type `struct ChanStruct` that is used to hold data relevant to inter-process communication, but now the `content` field is an entire array, rather than a single `unsigned int`. The functions in `ipc_intro.c` are updated to reflect reading and writing of locations within the `content` array rather than just a single `content` value. For example, the new `set_chan_content` is as follows.

```c
void set_chan_content(unsigned int chid, unsigned int index, unsigned int content)
{
    CHPOOL_LOC[chid].content[index]=content;
}
```

As before, there is a `content` parameter that we wish to save. However, now there is also an `index` parameter that tells us where in the `content` array (the second field of the `chid`th element of `CHPOOL_LOC`) to store it. Just as there is no check to make sure that `chid` is a valid index of `CHPOOL_LOC`, we do not check whether `index` is a valid index of `CHPOOL_LOC[chid].content`: that will likewise be covered in the front end (`ipc.c`). The rest of the functions in `ipc_intro.c` are similarly updated to reflect the use of an array to store data and can be seen in the Appendix. The proof of correctness of the back-end is covered below in a later section.
8.2 Front-End

The front-end functions (in `ipc.c`) must also be modified to reflect the use of a buffer to hold data for inter-process communication. Now instead of just checking that `chid` is a valid channel id, the front-end functions must also enforce that the buffer index that we are trying to write to or read from is valid, i.e. that it is less than `IPCBUFSIZE`. Below is the new implementation of the function `send`.

```c
unsigned int send(unsigned int chid, unsigned int index, unsigned int content)
{
    unsigned int info;
    if (0 <= chid && chid < NUM_CHAN &&
        0 <= index && index < IPCBUFSIZE) {
        info = get_chan_info(chid);
        if (info == 0) {
            set_chan_info(chid, 1);
            set_chan_content(chid, index, content);
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}
```

It is the same as before, except now in addition to making sure that `chid` is within the legal range (i.e. less than `NUM_CHAN`), it must also make sure that `index` is within its legal range (i.e. less than `IPCBUFSIZE`), and 0 is returned if either value is outside its respective legal range. When calling `set_chan_content`, `index` must be added to the parameters. The function `set_chan_info` is called as it was before, although possible further enhancements concerning the channel info (stored as the `isbusy` field of `struct ChanStruct` in `ipc_intro.c`) are explored in the following section. The rest of the functions in `ipc.c` are similarly updated and can be found in the Appendix, and the proof of correctness is addressed in a later section.

9 Possible Future Implementations

One potential flaw of the new implementation is that the information stored in the `isbusy` field of the `struct ChanStruct` in `ipc_intro.c` is not as useful as it could be. As with the original implementation, it is set to 0 when it is ready to receive data and set to 1 after it has received data. However, it does not specify where the data has been stored, and a process could potentially read from a part of the buffer other than where the data was written. Therefore, a future enhancement would involve encoding information about which parts of the buffer have been written to. Allowing writes to any index of the `content` array of `ChanStruct` would require one bit for each array index, but we could
more easily keep track of which indices have been written to if we enforce writing and reading from the beginning of the buffer. Then we would only need to keep track of the number of elements written rather than keep data for every single array index. In this case, preventing concurrency issues might require keeping another unsigned int to keep track of when a process is in the middle of reading to or writing from a buffer (although this information could also be encoded into the isbusy field).

Nevertheless, the above enhancements would not significantly change the proofs presented below. Notably, the implementation of the back-end would change very little, if at all, and while some parts of the front-end proof might change a little, the most difficult part of the proof (the buffer initialization), described below, would not change at all.

10 Proof of Correctness

Although the changes to the C source code are fairly simple, the changes to the Coq proofs of their correctness are significantly more involved. This demonstrates one of the difficulties in producing a certified operating system (or certified software of any kind), namely that the proof is often more time-intensive than the actual implementation itself, and that at times considerations of the proof’s difficulty may influence one’s choices when writing the source code. Below I outline parts of the proofs of the back-end (ipc_intro.c) and front-end (ipc.c). Full Coq proofs can be found in the code base available at the project web page: http://zoo.cs.yale.edu/classes/cs490/13-14b/whittaker.john.jlw24.

10.1 Proof of Back-End

The proofs involving setting and getting the channel info are (mostly) the same, the only difference being that the isbusy fields are at a different offset from the beginning of the CHPOOL_LOC array, since the intervening content fields are not single (4-byte) unsigned ints but an array of unsigned ints. The functions involving setting and getting values from a content buffer are a little more interesting, because they involve dealing with elements inside an array (content), which is a field inside a struct (a struct ChanStruct), which is itself inside an array of struct ChanStructs (called CHPOOL_LOC). Let us consider the function set_chan_content, reproduced below.

```c
struct ChanStruct {
    unsigned int isbusy;
    unsigned int content[IPCBUFSIZE];
};
struct ChanStruct CHPOOL_LOC[NUM_CHAN];

void set_chan_content(unsigned int chid, unsigned int content) {
    CHPOOL_LOC[chid].content = content;
}
```

Theorem 10.1. Let m be the initial memory state and le be the initial local environment. If the following assumptions hold
1. in le, chid, index, and content evaluate to $c_1$, $c_2$, and $c_3$, respectively;
2. the array \texttt{CHPOOL\_LOC} begins at location $l_1$;
3. $0 \leq c_1 < \text{NUM\_CHAN}$;
4. $0 \leq c_2 < \text{IPCBUFSIZE}$;

then the result of the executing the body of function \texttt{set\_chan\_content} will be termination, with a final memory state equal to the original state, except with $c_3$ (the value of content) stored at location $l_1 + 4 \ast (1+\text{IPCBUFSIZE}) \ast c_1 + 4 \ast (c_2 + 1)$ (corresponding to \texttt{CHANPOOL\_LOC}\[c_1\].content\[c_2\]).

\textbf{Proof.} From assumption 3 that $c_1$ (the value of chid) is within the legal range, it follows that \texttt{CHPOOL\_LOC}[c_1] is a valid struct ChanStruct at location $l_1 + c_1 \ast n$, where n is the size of struct ChanStruct. Each struct ChanStruct consists of 1 unsigned int for the isbusy field and an array of IPCBUFSIZE unsigned ints for the content field, for a total of 1+IPCBUFSIZE unsigned ints, each of which is 4 bytes wide. Therefore \texttt{CHPOOL\_LOC}[c_1] is a struct ChanStruct at location $l_1 + 4 \ast (1 + \text{IPCBUFSIZE} \ast c_1)$. Let us make $l_2 = l_1 + 4 \ast (1 + \text{IPCBUFSIZE} \ast c_1)$. Since \texttt{CHPOOL\_LOC}[c_1] is a valid struct ChanStruct, \texttt{CHPOOL\_LOC}[c_1].content is a valid unsigned int array of size IPCBUFSIZE at location $l_2 + \text{sizeof}($\texttt{CHPOOL\_LOC}[c_1].isbusy$)$. Since field isbusy is a single unsigned int, it is of size 4 bytes, resulting in a location of $l_2 + 4$. Let us make $l_3 = l_2 + 4$. From assumption 4 that $c_2$ (the value of index) is within the legal range and since \texttt{CHPOOL\_LOC}[c_1].content is a valid unsigned int array, it follows that \texttt{CHPOOL\_LOC}[c_1].content[c_2] is a valid unsigned int location of $l_3 + 4 \ast c_2$, since $c_2$ is the number of 4-byte-wide unsigned ints that come before \texttt{CHPOOL\_LOC}[c_1]. Let us make $l_4 = l_3 + 4 \ast c_2$. The C language’s = operator saves $c_3$ into location $l_4$, which unfolded is equal to $l_1 + 4 \ast (1+\text{IPCBUFSIZE}) \ast c_1 + 4 + 4 \ast c_2$, or equivalently $l_1 + 4 \ast (1+\text{IPCBUFSIZE}) \ast c_1 + 4 \ast (c_2 + 1)$.

The other proofs are of similar logic and can be seen in PThreadCode.v in the code base at http://zoo.cs.yale.edu/classes/cs490/13-14b/whittaker.john.jlw24.

10.2 Proof of Front-End

The front-end proofs are a little more involved than the back-end ones and required more changes from the proofs for the original implementation. In particular, the proof for \texttt{proc\_init} has become more complex. It originally used a while loop to initialize the isbusy and content fields of all the struct ChanStructs in the IPCPOOL\_LOC array, and in the new implementation it uses a nested while loop, the outer one to loop over every struct ChanStruct in IPCPOOL\_LOC and the inner one to loop over every element in the content array of struct ChanStruct in order to zero the isbusy fields (which can be done in the body of the outer loop) and every element of the content arrays (which is done in the inner loop). The function \texttt{proc\_init} uses calls to \texttt{set\_chan\_info} and \texttt{set\_chan\_content} in order to initialize values to 0. It is the most difficult of the front-end proofs and includes most of the interesting aspects of the other proofs. The function \texttt{proc\_init} follows below.
void proc_init(unsigned int mbi_addr)
{
    unsigned int i,j;
    vmcb_init(mbi_addr);
    i = 0;
    while (i < NUM_CHAN) {
        j = 0;
        set_chan_info(i,0);
        while (j < IPCBUFSIZE) {
            set_chan_content(i,j,0);
            j++;
        }
        i++;
    }
}

The call to vmcb_init is for initializing the Virtual Machine Code Block and
is not particularly relevant to the proof that follows. The heart of the proof
involves in reasoning about the while loops, preconditions that hold before en-
tering the loops, invariants that hold at the point of testing the loop invariant,
and postconditions that hold following the loop. First, let us consider the inner
loop, which sets every element in the content array of a struct ChanStruct
to 0 and increments j.

    while (j < IPCBUFSIZE) {
        set_chan_content(i,j,0);
        j++;
    }

Lemma 10.2. Given the preconditions (P), the inner loop will terminate, with
the invariants (I) holding at the point of testing the loop condition and the
postconditions (Q) holding after termination of the loop.

- P (preconditions):
  1. the memory state is some minit
  2. the local environment is some leinit
  3. in leinit, i evaluates to ival
  4. 0 ≤ ival < NUM_CHAN
  5. in leinit, j evaluates to 0
  6. in minit, stored at the location corresponding to IPCPOOL_LOC[ival]
     is a struct ChanStruct
  7. in minit, stored at the location corresponding to IPCPOOL_LOC[ival].isbusy
     is the unsigned int 0

- Q (postconditions):
  1. the local environment le is equivalent to leinit, except that j is set to
     IPCBUFSIZE
2. The memory state \( m \) is equivalent to \( \text{minit} \), with the exception that \( \forall j' \), 
\[ 0 \leq j' < \text{IPCBUFSIZE} \] at the location corresponding to \( \text{IPCPOOL} \[iv\].content[j'] \) is stored the unsigned int 0.

- **I (loop invariants)** given current memory state \( m \) and local environment \( le \):

1. In the local environment \( le \), \( j \) evaluates to some \( jval \)
2. \( 0 \leq jval \leq \text{IPCBUFSIZE} \)
3. One of the following holds:
   - \( jval = 0 \) and \( m = \text{minit} \) and \( le = \text{leinit} \) OR
   - \( jval > 0 \) and the memory state \( m \) is equivalent to \( \text{minit} \), with the exception that \( \forall j' \), \( 0 \leq j' < jval \) at the location corresponding to \( \text{IPCPOOL} \[iv\].content[j'] \) is stored the unsigned int 0 and the local environment \( le \) is equivalent to \( \text{leinit} \), except that \( j \) is set to \( jval \)

**Proof.** Let us first consider the question of whether the loop terminates. From P.5, the initial value of \( j \) is 0. We have proven that \text{set\_chan\_content} terminates. The statement \( j++ \) also terminates and is the only statement within the loop body that changes the value of \( j \). Therefore, the value of \( j \) increases by 1 in every execution of the loop body, and the loop will terminate when the value of \( j \) is equal to \( \text{IPCBUFSIZE} \), i.e. after \( \text{IPCBUFSIZE} \) executions of the loop body.

Now let us consider whether the three loop invariants hold at the top of the **while** loop, i.e. at the point of evaluating the loop condition. The first time we are at the top of the loop, the preconditions hold from our assumption. The loop invariants follow from the preconditions: I.1 (\( j \) evaluates to some \( jval \)) follows from P.5 since \( j \) evaluates to 0; I.2 follows likewise follows from P.5 since \( jval = 0 \), so \( 0 \leq jval < \text{IPCBUFSIZE} \); I.3 follows from P.1, P.2, and P.5 (simply the initial values of the memory state, local environment, and \( j \), respectively).

Next let us consider the case of a subsequent iteration of the while loop. In this case, we have \( jval \neq 0 \) since \( jval = 0 \) initially, and its value increases by 1 each time through the loop. We know, therefore, that \( 1 \leq jval \leq \text{IPCBUFSIZE} \) since if \( jval \) were greater than \( \text{IPCBUFSIZE} \), the previous test of the loop condition would have resulted in the termination of the loop.

Therefore, I.1 and I.2 hold. For I.3, we know \( jval > 0 \), and we also know that the local environment \( le \) is equivalent to \( \text{leinit} \), with the exception of \( j \) being equal to \( jval \), since nothing else in the local environment (i.e. \( i \)) is modified within the inner loop body. As for the memory state \( m \), we can use induction, with the base case being \( m = \text{minit} \) when \( jval = 0 \) (proven above). Therefore, if we assume that after when \( j \) evaluates to \( jval \), \( \forall j', 0 \leq j' < jval, \text{IPCPOOL} \[iv\].content[j'] = 0 \), then when \( j \) evaluates to \( jval + 1 \) (the following iteration), \( \forall j', 0 \leq j' < jval + 1, \text{IPCPOOL} \[iv\].content[j'] = 0 \). This is true because the call to \text{set\_chan\_content} sets the value at \( \text{IPCPOOL} \[iv\].content[jval] \) to 0 (the index is \( jval \) because the call occurs before the incrementing of \( j \)). From the induction hypothesis, we know that at indices 0 to \( jval - 1 \), \( \text{IPCPOOL} \[iv\].content \) has a value of 0, and nothing in the inner loop modifies those values. Thus, I.3 holds.

Finally, the postconditions \( Q \) directly follow from the loop invariant when \( jval = \text{IPCBUFSIZE} \). From the proof of termination, we know that the loop
terminates when jval = IPCBUFSIZE, so from that fact and I.1, Q.1 holds. And Q.2 likewise follows from the fact that jval = IPCBUFSIZE and the jval > 0 case of I.3.

Next, let us consider the outer loop, which sets j to 0, sets the isbusy field of the i-th element of IPCPOOL_LOC to 0 by calling set_chan_info, uses the inner loop to set all the elements of the content array of the same channel to 0, and then increments i.

```c
while (i < NUM_CHAN) {
    j = 0;
    set_chan_info(i,0);
    ... inner loop ...
    i++;
}
```

**Lemma 10.3.** Given the preconditions (P), the inner loop will terminate, with the invariants (I) holding at the point of testing the loop condition and the postconditions (Q) holding after termination of the loop.

- **P (preconditions):**
  1. the memory state is some minit
  2. in minit, stored at the location IPCPOOL_LOC is an array of struct ChanStruct of length NUM_CHAN
  3. in the local environment le, i evaluates to 0

- **Q (postconditions):**
  1. the memory state m is equivalent to minit, with the exception that
     - ∀ i′, 0 ≤ i′ < NUM_CHAN,
       - at the location corresponding to IPCPOOL_LOC.[i′].isbusy is stored the unsigned int 0 and
       - ∀ j′, 0 ≤ j′ < IPCBUFSIZE at the location corresponding to IPCPOOL_LOC.[i′].content[j′] is stored the unsigned int 0.

- **I (invariants):**
  1. in the local environment le, i evaluates to some ival
  2. 0 ≤ ival ≤ NUM_CHAN
  3. in minit, stored at the location IPCPOOL_LOC is an array of struct ChanStruct of length NUM_CHAN
  4. One of the following hold
     - ival = 0 and m = minit OR
     - ival > 0 and the memory state m is equivalent to minit, with the exception that ∀ i′, 0 ≤ i′ < ival,
       * at the location corresponding to IPCPOOL_LOC.[i′].isbusy is stored the unsigned int 0 and
∀ j', 0 ≤ j' < IPCBUFSIZE at the location corresponding to IPCPOOL_LOC[i'].content[j'] is stored the unsigned int 0.

Proof. By the same logic used for the inner loop, the outer loop also terminates because i is initialized to 0 and the only statement in the loop body that modifies the value of i is i++. The statements j = 0 and i++ terminate, as does the inner loop, proven above. Also, set_chan_info terminates, the proof of which can be found in file PThreadCode.v of the code base on the project website. Therefore, the value of i increases by 1 every loop iteration, and the loop will terminate when the value of i is NUM_CHAN.

Let us consider whether the loop invariants (I) hold, given the preconditions (P). First, let us consider the case upon first entering the loop. By P.3, i evaluates to 0, satisfying I.1 and I.2. I.3 directly follows from P.2, and the ival = 0 case of I.4 follows from P.1 and P.3.

Next let us consider the case of subsequent iterations through the loop. Since i is incremented during every loop variation, it must be that ival > 0. Also, it must be the case that ival ≤ NUM_CHAN since if it were any greater, the loop would have terminated on a previous iteration. Therefore I.1 and I.2 hold. I.3 also holds because none of the statements inside the loop affect the allocation of memory for the IPCPOOL_LOC array. I.4 can be shown through induction, with the base case of ival = 0 holding trivially. For the ival > 0 case, we assume that ival > 0 and that the statements about memory hold ∀i′, 0 ≤ i′ < ival. For ival + 1, we know ival + 1 > 0 and that IPCPOOL_LOC.[ival].isbusy has been set to 0 from the proof of set_chan_info. Now to reason about the inner loop, we must show the preconditions we used for the previous lemma hold.

We use the current memory state and local environment as minit and leinit (Pinner.1 and Pinner.2). Pinner.3 and Pinner.4 follow from I.1 and I.2 shown earlier, since i is not incremented until after the inner loop so ival < NUM_CHAN. Pinner.5 holds since we set j to 0 before executing the inner loop. Pinner.6 holds from I.3, shown earlier, and since ival is a legal index of IPCPOOL_LOC from Pinner.4. Pinner.7 holds from the execution of set_chan_info, as stated earlier. Since all the preconditions of the inner loop hold before the execution of the loop, the postconditions hold after, so from Qinner.2 ∀j′, 0 ≤ j′ < IPCBUFSIZE at the location corresponding to IPCPOOL_LOC[i'].content[j'] is stored the unsigned int 0. This also holds true for values less than ival from the induction hypothesis, so I.4 holds.

Finally, the postcondition Q.1 holds directly from the ival > 0 case of I.4 since as shown in the proof for termination, i will evaluate to NUM_CHAN.

Theorem 10.4. Now that we can consider the loop as a unit, given the preconditions are satisfied, we can prove that the function body as a whole terminates and that ∀i′, 0 ≤ i′ < NUM_CHAN the isbusy field of IPCPOOL_LOC[i′] is set to zero and ∀j′, 0 ≤ j′ < IPCBUFSIZE content[j′] is also set to zero.

```c
void proc_init(unsigned int mbi_addr)
{
    unsigned int i, j;
    vmcb_init(mbi_addr);
    i = 0;
}  
```
Proof. The call to \texttt{vmcb_init} terminates, as proven in \texttt{VVMCBIntroCode.v} (this is no different from the original implementation of \texttt{ipc.c}). Now, we must show that the preconditions of the outer loop hold. Let the current memory state be called \texttt{minit} (P.1). P.2 is satisfied by the definition of \texttt{IPCPOOL\_LOC}, defined in \texttt{ipc\_intro.c}, and P.3 holds because \texttt{i} is set to 0 immediately before entering the loop. Since the preconditions hold entering the loop, the postconditions hold upon exiting, giving us the \texttt{IPCPOOL\_LOC} array containing \texttt{struct ChanStruct} with zeroed \texttt{isbusy} fields and \texttt{content} arrays with elements set to 0.

The full proof, formalized in Coq, can be found in \texttt{PIPCIntroCode.v}. The rest of the front-end proofs are similar but simpler since only \texttt{proc\_init} contains \texttt{while} loops. Auxiliary lemmas and data structures used to formalize and simplify the loop invariants and postconditions can be found in \texttt{CInitSpecsproc.v}, and the formalization of \texttt{struct ChanStruct} can be found in \texttt{CDataTypes.v}.

11 Conclusion

As the recent Hearbleed Bug of OpenSSL [13] and the “goto fail” bug of Apple’s SSL [15] demonstrate, even fairly straightforward and dangerous bugs can go unnoticed for great lengths of time. Especially in case of communication over the Internet or with other processes, we cannot assume that the intentions of other agents are benign. This is why for critical processes—military, medical, financial, or otherwise—certified software is coming into greater and greater demand. In this paper, I have presented a blueprint for safe and convenient inter-process communication. Certainly, this is hardly the last word on certified inter-process communication, but I hope that it can serve as a good introduction for those who might later build upon this work or work on other forms of certified software. The fact that proofs, in this case on properties of programs, can be of use not just in academia but also in industry might come as a surprise to some. Proofs, although still imperfect in some instances, provide an additional layer of assurance beyond human intuition and thorough testing. A commercially viable, fully certified operating system would make the computer-centered processes in our lives safer and more reliable, potentially revolutionizing not just computing but the economy as well.

12 Acknowledgments

I would like to thank Prof. Zhong Shao of the Department of Computer Science for taking the time to advise me and for allowing me access to his research and his team members. Prof. Shao’s Formal Semantics and Compilers classes were two of the most enjoyable and edifying courses that I have taken in my time at Yale, and the former provided me with the tools necessary to embark on this project.

I would also like to thank Xiongnan (Newman) Wu, a Ph.D. candidate in the Department of Computer Science, for the incredible help and support he provided me during the semester. He taught me the ins and outs of Coq and
CertiKOS and gave me insight into the beauty and simplicity of computer-assisted proof techniques.

References


