Formally verifying Exceptions for Low-level code with Separation Logic

Marco Paviotti[a], Jesper Bengtson[b]

[a]*School of Computing, University of Kent, Canterbury, United Kingdom*
[b]*IT University of Copenhagen, Copenhagen, Denmark*

Abstract

Exceptions in low-level architectures are implemented as synchronous interrupts: upon the execution of a faulty instruction the processor jumps to a piece of code that handles the error. Previous work has shown that assembly programs can be written, verified and run using higher-order separation logic [14]. However, execution of faulty instructions is then specified as either being undefined or terminating with an error. In this paper, we study synchronous interrupts and show their usefulness by implementing a memory allocator. This shows that it is indeed possible to write positive specifications of programs that fault. All of our results are mechanised in the interactive proof assistant Coq.

Keywords: Formal Verification, Separation Logic, Assembly, Coq, Exceptions, Step-indexed models, Interactive Theorem Proving

1. Introduction

Assembly code is difficult to prove correct. When verifying imperative programs, standard Hoare-logics often make implicit assumptions about the control flow of programs and assume that the code c in a triple $\{P\}c\{Q\}$ has one entry point and one exit point, even though it may internally contain loops and method calls. In assembly programs we cannot make this assumption as the control flows of these languages are inherently unstructured.

Control flow is altered primarily by two mechanisms – jump commands and interrupts. Jump commands allow developers to execute code stored nearly

anywhere in memory; their use is an active choice, much like writing a loop or calling a method. Interrupts, on the other hand, occur either when something has gone catastrophically wrong (such as dividing by zero or reading from unmapped memory) or when an action from the environment requires processing (such as the user pressing a key, a change to the file system is made, or the processor clock ticks).

While some aspects of interrupts might resemble those of function calls, there are substantial differences: synchronous interrupts are not called explicitly, but trigger as a result of a particular operation on a particular state, e.g. division by zero. Interrupts that trigger as a result of an error are typically referred to as synchronous, while asynchronous interrupts are external requests. Another name for synchronous interrupts is exceptions, due to their similarity with the exceptions encountered in languages like Java or ML, and we will use the terms interchangeably.

We build on the existing Coq [22] formalisation of the x86 instruction set [11] by Jensen et al. [15]. The memory model (explained in Section 3.1) is very close to that of the actual x86 chipset – control flow is implemented using jumps which are inherently unstructured and code is stored in memory. This allows for self-modifying code. Secondly, their program logic [14] is able to handle non-structured control flow through jumps in a clean and concise manner (explained in Section 3.2).

In this paper, we present a monadic semantics and a program logic to verify x86-assembly programs that feature *synchronous* interrupts. Although we cannot ensure verified assembly can be run on real hardware as in the previous formalisation, we are able to model synchronous interrupts very closely to the way they run on real processors.

Whenever an interrupt fires – be it synchronous or asynchronous – the machine jumps to a piece of code to handle the event. On x86 architectures, the machine operates by using an Interrupt Description Table (IDT). The IDT is stored in memory and contains, for every type of interrupt, a pointer to the handler for that interrupt. When an interrupt is fired, the processor's state

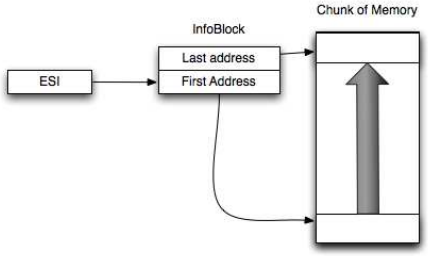
is saved, the address of the interrupt handler is retrieved from the table and the code of the handler is executed. Barring catastrophic failure, the original processor state is then typically restored and its execution is resumed. Similarly to how Jensen et al. store code in memory, we store the IDT and the interrupt handlers in memory, which opens up the possibility for programs to update the various handlers dynamically.

In this work, we initiate an exploration of the formalisation of interrupts for assembly machines. We do this by formalising *synchronous interrupts*. We believe the experience gained here gives valuable insights for extending this work to asynchronous interrupts, and this discussed further in Section 7.

Our contributions are the following.

- We extend the semantics of Jensen et al. to support synchronous interrupts, (Section 4.1). Jensen's model is formalised under the assumption that the machine runs in Protected Mode and we follow their design decisions.
- We add rules to the program logic to cover cases where synchronous interrupts are thrown, for example when reading from unmapped memory (Section 4.2), allowing users to verify programs that use interrupts.
- We verify a small memory allocator that uses synchronous interrupts (Section 5). To do this, some technical, but crucial, lemmas (Section 5.1) have been proven. (Section 5.2).
- All of our results are mechanised in Coq.

The source code for our mechanisation can be found at `http://www.itu.dk/people/mpav/downloads/coqdev.zip`. The additional work with respect to the previous development amounts to 1084 lines of code including the allocator, which comprises 182 lines of code. The work took approximately four months. The code is compiled with `coqc` version `8.4pl3` with `OCaml 4.00.1`.



$$\begin{aligned}\mathsf{allocImp}(\mathit{info}, n, \mathit{fail}) \triangleq\ & \mathsf{mov\ ESI}, \mathit{info};\\ & \mathsf{mov\ EDI}, [\mathsf{ESI}];\\ & \mathsf{add\ EDI}, n;\\ & \mathsf{jc}\ \mathit{fail};\\ & \mathsf{cmp}\ [\mathsf{ESI}+4],\ \mathsf{EDI};\\ & \mathsf{jc}\ \mathit{fail};\\ & \mathsf{mov}\ [\mathsf{ESI}],\ \mathsf{EDI}.\end{aligned}$$

Figure 1: Standard Allocation mechanism

2. Memory allocation using exceptions

We use the standard AT&T syntax for assembly notation. For this example, '$\mathsf{mov}\ r,\ v$' stores the value v in the register r, '$[r]$' dereferences a pointer stored in r, '$\mathsf{add}\ r,\ v$' adds the value v to the value stored in the register r – if the result to be stored in r exceeds the capacity of the register $(2^{32}-1)$ the carry flag is set. The instruction '$\mathsf{jc}\ a$' jumps to address a if the carry flag is set. Finally, '$\mathsf{cmp}\ r\ u$' compares the value stored in register r with the value stored in u and sets the carry flag if the former is greater than the latter.

Jensen et al. [14] implemented and verified a simple bound-and-check memory allocator, whose behavior and code are depicted in Figure 1. It takes three arguments, *info*, n and *fail*, where *info* is a pointer to an information block of two cells pointing to the beginning and the end of the storage respectively; n is the number of bytes to be allocated, and *fail* is an address that the allocator jumps to if n bytes are not available. The program does two comparisons – the first checks if adding n to the memory start address causes the register value to wrap around (by resulting in a value greater than $2^{32}-1$) and the second checks if that number is outside the memory available to the allocator. In both

cases, the allocator jumps to *fail* if the test succeeds.

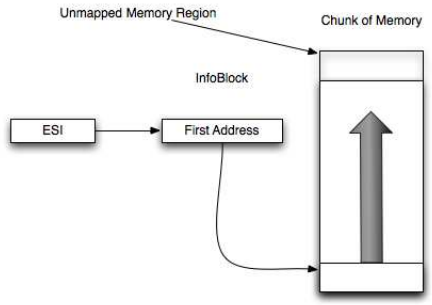
We verify an alternative version of this memory allocator using our new semantics and program logic for exceptions, whose behaviour and code are depicted in Figure 2. In our allocator we use the exception mechanism to jump to a handler in case of failure, thus there are no checks for overflow or memory bounds. Instead, we mark the end of the available memory with an unmapped location.

The unmapped memory can be implemented via the paging system of the Intel x86 machine, which is a mechanism to allow the machine to see more memory than is physically available. This is implemented using a virtual address space, namely a table with records pointing to the physical memory. A very useful additional feature of paging is the ability to mark a single page with security permissions. In this way, user processes can read and write only in their own allocated space, thus implementing process isolation at the hardware-level.

An operating system can choose to set up some virtual memory for a process and mark the end of it with a read-only memory cell. We implement an allocator that makes use of this strategy.

Our allocator has three arguments, v_1, v_2 and *info*. The first two are double word values to be written in the memory and the third is a pointer to the start of the memory. Thus, we allocate two double words and write the data at the same time. By virtue of this write to memory an exception is triggered whenever that memory is unmapped, i.e. when the end of the memory available to the allocator has been reached. It is then up to the interrupt handler to catch the exception, but by jumping to the *fail* address it will mimic the behaviour of the handler in Figure 1.

The reasons behind this design of the allocator are twofold. First, the memory needs to be 'touched' somehow in order to trigger the exception. If we chose not to write into the allocated space, as in Figure 1, we would allow a client to circumvent the exception mechanism as the pointer to the next cell might go beyond the unmapped memory region. and since this region is not touched the allocator will succeed. Secondly, we can only allow the user to allocate a



$$
\begin{aligned}
\mathsf{allocImpExp}\ v_1\ v_2\ \mathit{info} \triangleq\ & \mathsf{mov\ ESI},\ \mathit{info};; \\
& \mathsf{mov\ EDI},\ [\mathsf{ESI}];; \\
& \mathsf{mov}\ [\mathsf{EDI}],\ v_1;; \\
& \mathsf{add\ EDI},\ 4; \\
& \mathsf{mov}\ [\mathsf{EDI}],\ v_2; \\
& \mathsf{add\ EDI},\ 4; \\
& \mathsf{mov}\ [\mathsf{ESI}],\ \mathsf{EDI}.
\end{aligned}
$$

Figure 2: Allocation mechanism with exceptions

fixed number of cells. If we allowed an arbitrary number of cells, since this memory region has to be initialised as mentioned above, we would need to either generate a corresponding amount of writes into memory – which would render the allocator less efficient w.r.t. Figure 1 – or let the user pass on the data to be written in the store. In the latter case, the client would be already in possession of a chunk of memory containing the source data and thus, the allocator would be a mere memory-copy routine.

For our own purpose it suffices to allocate two cells, since this enables data structures such as linked lists, e.g. a client can store the address of its callback routine into the IDT and then use the allocator as a *cons* routine.

A variant of this allocator with any other fixed number of cells can be verified if needed.

2.1. Interrupt mechanism

Every interrupt is identified by a unique number which is an index into a table of pointers to the handlers. This table is commonly referred to as the *Interrupt Descriptor Table* (IDT) and it is a chunk of memory pointed to by the

IDT Register (IDTR) and further divided into records. Every record of the IDT contains a pointer to an interrupt handler.

In Intel Protected mode, when segmentation is used, every address is uniquely calculated by giving a pair of addresses called the *base* and *offset* respectively. For example, the pair CS:EIP is the address of a piece of code with offset EIP inside the segment indicated by the value of CS.

When an interrupt triggers, the CPU looks up the number of the interrupt, saves the values of CS, EIP and any flags to the stack, and jumps to the appropriate handler. In most operating systems, only one segment is used so we chose not to worry about the segment selector – in particular the code selector – as it would be easy to formalise if needed. The reader can safely skip this detail as we will not use it in this paper.

While the CPU is in charge of saving the return address to the stack and jumping to the handler, it is the responsibility of the programmer to implement a "safe" handler, i.e. one that leaves the machine in a state from which the original program can continue without faulting.

An interrupt handler is called *transparent* when it leaves no trace of its execution in memory, i.e. the memory looks the same before and after the interrupt fired. This kind of handler saves the CPU state by pushing all the registers to the stack, handles the interrupt, and then restores the state as it was before it was interrupted. Finally, it executes the IRET instruction (Return from Interrupt). This signals the CPU to restore the triple CS:EIP and FLAGS, thus performing a far jump back to where the program was interrupted.

The exception mechanism is slightly different in the presence of a faulty handler. If the handler produces an error the machine raises a *Double Fault* exception, which behaves the same as the other exceptions. Should this handler fail, the whole machine reboots. This situation is called *Triple Fault*.

In the following section we will present the semantics of this behaviour. However, we do make some simplifications such as avoiding mentioning segments. This is because, despite the fact that segmentation is a useful security mechanism, in many operating systems it is not used. Segmentation is achieved

by mapping every segment, be it code, data or stack segment, to the whole range if physical memory. Implementors argue that the segmentation system in the Intel's protected mode is very expensive and thus the old paging system is preferred. When an address is referred to inside the same segment the code is in, the Intel specification states that we can avoid mentioning it.

3. Assembly semantics and logic

3.1. Semantics

In this section we present the Coq semantics from Jensen et al. [14]. The semantics of the assembly language operates on a total state consisting of all registers, flags, and memory, as follows:

$$\begin{aligned}\mathbb{S} \triangleq\ & (\mathsf{reg} \to \mathsf{DWORD}) \times \\ & (\mathsf{flag} \to \{\mathsf{true}, \mathsf{false}, \mathsf{undef}\}) \times \\ & (\mathsf{DWORD} \to (\mathsf{BYTE} \uplus \{\mathsf{unmapped}\}))\end{aligned} \tag{1}$$

Here, $\mathbb{S}$ is made of three total functions: the register state, mapping each register to a DWORD (a 32-bit value); the flag state, mapping each flag to a boolean value or bottom; and the memory, mapping each 32-bit address to a BYTE (an 8-bit value), plus an unmapped value. The unmapped value stands for an inaccessible byte of memory. For example, it can be used when the memory is protected for some reason or inaccessible, e.g. some parts of the BIOS [11]. Let E be the set of numbers from 0 to 255. These represent the errors that can occur, e.g. Division By Zero, General Protection and so on.

The result of a computation is either an error in E, an *unspecified behavior* or a result of type X along with the updated state:

$$R_X \triangleq \mathsf{error}\ E \mid \mathsf{unspecified} \mid \mathsf{update}\ \mathbb{S}\ X$$

The semantics of the machine are monadic: programs are functions that takes a state $\mathbb{S}$ and produce a result. The type of a computation is the following:

$$ST\ X \triangleq \mathbb{S} \to R_X$$

ST is a *state monad* with the usual *return* ($\eta : A \to ST\ A$) and *bind* operations $\texttt{>>=} : ST\ A \to (A \to ST\ B) \to ST\ B$. We use $\mathsf{let}\ x \leftarrow c;c'$ for $c \texttt{>>=} \lambda x.c'$ and $\mathsf{do}\ c;c'$ for $c \texttt{>>=} \lambda\_.c'$. For each field of the state, we have read and write operations: $\mathsf{read}_{\mathsf{Flags}}$, $\mathsf{read}_{\mathsf{Reg}}$ and $\mathsf{read}_{\mathsf{Mem}}$ to read the value of flags, registers and memory locations respectively, and write operations $\mathsf{set}_{\mathsf{Flags}}$, $\mathsf{set}_{\mathsf{Reg}}$, $\mathsf{set}_{\mathsf{Mem}}$ to write to that state. Note, the monad is defined on a general type X. This is because we also need computations that return instructions and pointers, as described later in this section.

The instruction set Instr is inductively defined. The interpretation function maps an instruction *instr* onto an element $[\![intr]\!]$ of the monad $ST\ \mathsf{unit}$, i.e. a function that takes a state and returns the unit value along with the modified state or an error. For example the interpretation of the jmp instruction is defined as:

$$[\![\mathsf{jmp}\ i]\!] \triangleq \mathsf{set}_{\mathsf{Reg}}(\mathsf{EIP} := i)$$

That is, a jump instruction is a computation that updates the EIP register with the address specified by the instruction.

The semantics of the machine make use of a step function of type $ST\ \mathsf{unit}$ which fetches, decodes and executes an instruction as follows:

$$\begin{aligned}\mathsf{step} \triangleq\ & \mathsf{let}\ eip \leftarrow \mathsf{read}_{\mathsf{Reg}}(\mathsf{EIP});\\ & \mathsf{let}\ (instr, neweip) \leftarrow \mathsf{decode}(eip);\\ & \mathsf{do}\ \mathsf{set}_{\mathsf{Mem}}(\mathsf{EIP} := neweip); [\![instr]\!]\end{aligned}$$

where decode is a function that takes a 32-bit address and decodes the value it contains into an instruction and the new instruction pointer, or returns an error. If the decoding fails, the entire step-function fails with the same error.

The semantics are defined by means of a function run of type $\mathbb{N} \times ST\ \mathsf{unit} \to ST\ \mathsf{unit}$ together with the step function defined above. run is defined recursively on its first argument and takes as input a natural number n and a computation, and executes that computation for n steps.

3.2. Separation Logic for low-level code

In this section we give some background on the program logic we use for reasoning about low-level code [14].

A specification for the assembly language is given in continuation-passing style and has the following form:

$$\vdash (\mathsf{safe} \otimes \mathsf{EIP} \mapsto j * Q \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mapsto i * P) \oslash i..j \mapsto c \tag{2}$$

which states that a program c stored in the memory between the address i and address j is safe to run from a state P *provided* that there is a continuation that runs safely from Q, where P and Q are separation logic formulas [19, 20, 18, 13, 7, 6] and safe is predicate in the specification logic. The tensor operator $\otimes$ takes a specification S and an assertion formula P and forms a new specification $S \otimes P$. The read-only tensor operator $\oslash$ has the same type as $\otimes$, but it is used to compose specifications with chunks of memory that contain code that ought not to be modified.

The $\otimes$ operator is an invariant extension operator in the following sense

$$(\mathsf{safe} \otimes P \Rightarrow \mathsf{safe} \otimes Q) \oslash i..j \mapsto c \otimes R \dashv\vdash \mathsf{safe} \otimes (P * R) \Rightarrow \mathsf{safe} \otimes (Q * R) \oslash i..j \mapsto c$$

where R is the assertion that is preserved by the computation. This can be shown by distributing R over the implication and using the rule $S \otimes P \otimes R \dashv\vdash S \otimes (P * R)$.

If c is a block, the specification in (2) can be read as a standard Hoare triple $\vdash \{P\}c\{Q\}$ for partial correctness.

In the assertion logic, besides the separating conjunction $*$, there are points-to relations for registers and flags, $\mapsto$, and for the memory, $\mapsto$. As code is data, code can be specified in the assertion logic by using the $\mapsto$ relation, e.g. $i..j \mapsto c$ means the memory between i and j contains the program c. We are going to use the question mark $r?$, where r is a register, as syntactic sugar for $\exists v, r \mapsto v$ and similarly for the memory. Separation logic entailments are solved conveniently within `Charge!` [3, 4] – a library of Coq tactics for program verification.

As an example, the mov instruction can be specified using the rule format

(2) by instantiating as follows:

$$\begin{aligned} Q &\triangleq r_1 \mapsto_{\triangleright} pd * pd \mapsto v_2 * r_2 \mapsto_{\triangleright} v_2 \\ P &\triangleq r_1 \mapsto_{\triangleright} pd * pd \mapsto v_1 * r_2 \mapsto_{\triangleright} v_2 \\ c &\triangleq \mathsf{mov}\ [r_1], r_2 \end{aligned} \tag{3}$$

If the machine is safe to run from a state where EIP points to j, r_1 is a register containing a memory region that contains v_2 and r_2 is a register containing the value v_2, then the machine is safe to execute a mov instruction located at address i with register r_1 pointing to a different value.

Another example is the assembly program that sits in a tight loop forever:

$$\vdash \mathsf{safe} \otimes \mathsf{EIP} \mapsto_{\triangleright} i \oslash i..j \mapsto \mathsf{jmp}\ i \tag{4}$$

In order to prove this statement, however, we need to prove the same statement after one step of computation, i.e. after the jump has been made. A convenient way to express this inside the logic is to use the $\triangleright$ modality, pronounced "later". To prove the statement above it suffices to prove that if it holds "later" then it holds now. This proof technique goes under the name of *Löb Induction* [1, 17, 5].

Finally, it is possible to compose specifications of programs in a modular way. A program is composed by one instruction and another program as follows:

$$\frac{\begin{array}{l} \vdash (\mathsf{safe} \otimes \mathsf{EIP} \mapsto_{\triangleright} i_1 * Q_1 \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mapsto i * P_1) \oslash i..i_1 \mapsto c_1 \\ P \vdash P_1 * R_{\mathrm{P}} \\ \vdash (\mathsf{safe} \otimes \mathsf{EIP} \mapsto j * Q \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mapsto_{\triangleright} i_1 * Q_1 \otimes R_{\mathrm{P}}) \oslash i_1..j \mapsto c_1; c \end{array}}{\vdash (\mathsf{safe} \otimes \mathsf{EIP} \mapsto j * Q \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mapsto_{\triangleright} i * P) \oslash i..j \mapsto c_1; c} \tag{5}$$

The code snippet $c_1; c$ is the composition of the first instruction c_1 and the rest of the program c. In order to prove the whole program is safe to start from the beginning with memory in P we have to prove that P satisfies the precondition required by the first instruction c_1. Secondly, we have to prove the rest of the program is safe to execute from the part of the memory modified, namely Q_1 and part of P left untouched, namely R_P. Note that the code part of the specification is left unchanged. What changes is the program pointer to the memory where the code lies.

3.3. Memory representation

As mentioned in the introduction our model differs from the x86 in terms of a small implementation detail that means our formalisation is not runnable on real hardware. However, we want to stay as close as possible to the original formalisation which allowed a user to write actual x86 assembly code, verify it, extract it as machine code and run it. To do this we chose to inherit the original memory representation where values are encoded as vectors of booleans (lists of a set length), representing binary numbers.

```
Definition BITS (n: nat) := list n bool.
```

here n is the length of the list. Double words are defined similarly, using the previous definition, as a list of 32 bits:

```
Definition DWORD := BITS 32.
```

The definition of these types and their functions use modulo 2^n arithmetic in Coq. This is not suitable for points-to predicates in separation logic. Consider the predicate $p..q \mapsto v$ for p and q of type DWORD. Assuming this predicate it is not possible to infer that $p \leq q$ in arithmetic modulo 2^{32} as $p + 4$ might wrap around. A work-around consists in defining an additional type dependent on n that adds a `top` element representing the end of the memory:

```
Inductive Cursor (n: nat) := mkCursor (p: BITS n) | top.
```

A cursor is either a list of bits or the memory beyond the last representable address.

This representation of bit values pervades the whole framework and some challenges arise when trying to reason about points-to relations, particularly when reasoning about lists of memory cells. We defer this issue to Section 5.1.

4. Semantics and logic for exceptions

4.1. Semantics for exceptions

We lift the semantics from Section 3.1 to model the exceptions. We do not expect that any useful handler could throw an exception, but for the sake of

completeness we define the semantics such that when an exception is thrown inside a handler a double fault triggers. If this happens inside the double fault handler the machine reboots (triple fault).

First, we define *throwexp* which takes as input the interrupt level and the number of the interrupt to throw.

$$
\begin{aligned}
\mathit{throwexp}(\mathit{intl}, n) \triangleq \quad & \mathsf{do}\ \mathsf{set}_{\mathsf{Reg}}\mathsf{INTL} := (\mathit{intl} + 1); \\
& \mathsf{let}\ \mathit{idt} \leftarrow \mathsf{read}_{\mathsf{Reg}}(\mathsf{IDTR}); \\
& \mathsf{let}\ \mathit{eip} \leftarrow \mathsf{read}_{\mathsf{Reg}}(\mathsf{EIP}); \\
& \mathsf{do}\ \mathsf{push}(\mathit{flags}); \\
& \mathsf{do}\ \mathsf{push}(\mathsf{EIP}); \\
& \mathsf{let}\ \mathit{new} \leftarrow \mathsf{read}_{\mathsf{Mem}}(\mathit{idt} + n * 4); \\
& \mathsf{set}_{\mathsf{Reg}}(\mathsf{EIP} := \mathit{new})
\end{aligned}
$$

This routine works as follows. The semantics looks up the address of the IDT by reading it from the IDTR and then saves the address and flags of the current execution point – denoted by $\mathsf{push}(\mathit{eip})$ and $\mathsf{push}(\mathit{flags})$ – by pushing them to the stack pointed to by ESP İt then fetches the address of the corresponding exception by looking up the value of the nth record inside the IDT, and saves this value (the address of the interrupt handler) to the EIP register.

We next define the double fault exception as a particular case of the former, namely an exception that occurs at interrupt level 1 and that throws a double fault exception.

$$\mathit{doublefault} \triangleq \mathit{throwexp}\ 1\ \mathit{ExnDF}$$

Finally, we define the *throw* function which throws the exception number n.

$$
\begin{aligned}
\mathit{throw}(n : \mathtt{nat}) \triangleq\ & \mathsf{let}\ \mathit{level} \leftarrow \mathsf{read}_{\mathsf{Reg}}\mathsf{INTL}; \\
& \mathsf{if}\ (\mathit{level}{=}0)\ \mathsf{then}\ \mathit{throwexp}\ \mathit{intl}\ n \\
& \mathsf{else}\ \mathsf{if}(\mathit{level}{=}1)\ \mathsf{then}\ \mathit{doublefault} \\
& \mathsf{else}\ \mathit{halt}
\end{aligned}
$$

The above routine can be read as follows: if the INTL register is zero the machine raises it to one and throws the corresponding exception. If the INTL register is one, we raise a double fault exception. Otherwise, we are in the third level of interruptions and the machine halts.

In order to catch the error and throw an exception we define a catch function of type $\mathsf{catch} : ST\ \mathsf{unit} \to ST\ \mathsf{unit}$ which takes a computation and gives a computation such that if the former ends up in a fault it throws an exception and otherwise returns the same result:

$$
\begin{aligned}
\mathsf{catch}\ (c : ST\ \mathsf{unit})(s : \mathbb{S}) \triangleq\ & \mathsf{case}\ (c\ s)\ \mathsf{of} \\
& \mid \mathsf{error}(n) \Rightarrow \mathit{throw}(n) \\
& \mid x \Rightarrow x \\
& \mathsf{end}
\end{aligned}
$$

Whenever we have a computation c, we obtain a computation $\mathsf{catch}\ c$ of type $ST\ \mathsf{unit}$ which turns errors into exceptions. We can then use the catch function with the interpretation function for instructions from Section 3.1: if $\mathit{instr} \in \mathsf{Instr}$ then $\mathsf{catch}(\llbracket \mathit{instr} \rrbracket)$ is the computation that throws an exception whenever the instruction *instr* fails to execute.

We substitute this term in the definition of step as follows:

$$
\begin{aligned}
\mathsf{step} \triangleq\ & \mathsf{let}\ \mathit{eip} \leftarrow \mathsf{read}_{\mathsf{Reg}}(\mathsf{EIP}); \\
& \mathsf{let}\ (\mathit{instr}, \mathit{neweip}) \leftarrow \mathsf{decode}(\mathit{eip}); \\
& \mathsf{do}\ \mathsf{set}_{\mathsf{Mem}}(\mathsf{EIP} := \mathit{neweip}); \mathsf{catch}(\llbracket \mathit{instr} \rrbracket)
\end{aligned}
$$

Note that the *only* difference between this step function and the one from the previous section is in the final command where we change $[\![instr]\!]$ to $\mathsf{catch}([\![instr]\!])$.

Whenever the instruction just fetched from the memory raises an exception, the machine jumps to the respective handler by updating the EIP register and the machine continues executing from there.

We could have modified each instruction to throw an exception instead of generating an error and then trapping it. This would result in a simpler monadic semantics. However, our approach is essentially equivalent and it makes sure pre-existenting code is left untouched thus remaining faithful to the previous formalisation.

4.2. Specification logic for exceptions

In order to be able to reason about these exceptions we need assertions describing the state of the memory in which these events can be triggered. Such assertion allows reasoning about read and write operation to unmapped memory regions. We define a predicate $l \mapsto !!$ that denotes the set of states such that the location l maps to an unmapped location.

Every instruction that tries to read or write from an unmapped location now becomes a jump into the exception handler, provided that the IDT is present in memory.

We present the additional rule for a mov instruction that accesses an unmapped memory region. Thus, we make a jump-like variant of (3) using the later-operator. The rule has the following shape:

$$\vdash (\triangleright\mathsf{safe} \otimes Q \Rightarrow \mathsf{safe} \otimes P) \oslash i..j \mapsto \mathsf{mov}\ [r_1], c \otimes \mathsf{Inv} \tag{6}$$

where P is a precondition that states the code will run from a state where r_1 points to an unmapped memory location and where no interrupts have occurred; Q is the postcondition *after one step*, describing the state after the interrupt triggered as a result of reading the memory pointed to by r_1; and where Inv contains the IDT table. Informally, if the machine is "later" safe to run from

inside the handler, where the interrupt level is set to 1 and the stack pointer points to the top of the stack where the return address j is stored, then it is safe to run from a state where the interrupt level is zero and the stack pointer points to a cell of memory such that the next cell is mapped.

First, the precondition must ensure that the program is starting from i with interrupt level 0 and with the necessary space in the stack to allocate the return address:

$$P \triangleq (\mathsf{EIP} \mapsto i * \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto sp * ((sp-4)..sp)?) \tag{7}$$

Secondly, we need the post-condition to assert that it is safe to jump into the handler. Thus, at this point in time the program pointer will be the $fail$ address with interrupt level set to 1 and the return address on top of the stack:

$$Q \triangleq \mathsf{EIP} \mapsto fail * \mathsf{INTL} \mapsto 1 * \mathsf{ESP} \mapsto (sp-4) * (sp-4)..sp \mapsto j \tag{8}$$

Finally, we need an invariant asserting that the IDT is in the memory. For a natural number n and a 32-bit address y, we denote by $\mathsf{IDT}[n/y]$ the IDT in which the nth record contains the address y. Thus, the following definition, $\mathsf{IDT}[\mathsf{ExnGP}/\mathsf{fail}]$ stands for the IDT table where the record associated with the general protection exception is updated with the address fail, namely, the address to the handler. Moreover, the location l points to the unmapped region and more precisely, the bytes from l to $l+4$ (excluding $l+4$ itselft) are unmapped, while r_1 points to l. Note that even though a complete IDT contains pointers to all handlers we do not need all of them, just the pointer to the handler associated with the general protection exception.

$$\mathsf{Inv} \triangleq (r_1 \mapsto l * l..(l{+}4) \mapsto !! * \mathsf{IDT}[\mathsf{ExnGP}/fail]) \tag{9}$$

Any instruction that makes use of an unmapped cell turns into a jump. In the Coq development we proved similar rules for the read version of mov and for push and pop instructions.

We proved that the push instruction is safe to execute when the parameter

refers to an unmapped location:

$$
\begin{array}{ll}
\vdash & \forall i : \mathsf{DWORD}, j : \mathsf{DWORD}. \\
& (\triangleright(\mathsf{safe} \otimes (\mathsf{EIP} \mapsto \mathsf{fail} * \mathsf{INTL} \mapsto 1 * \mathsf{ESP} \mapsto (sp{-}4) * (sp{-}4)..sp \mapsto j)) \Rightarrow \\
& \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto sp * (sp{-}4)..sp \mapsto v)) \\
& \quad \oslash(i..j \mapsto \mathsf{push}[r]) \\
& \quad \otimes(r \mapsto l * l..(l{+}4) \mapsto !! * \mathsf{IDT}[\mathsf{ExnGP}/\mathsf{fail}])
\end{array}
\tag{10}
$$

The structure of this rule is the same as in rule (6), but here we *read* from an unmapped memory address and the destination is the top of the stack.

It may be worth noting that a variant of this rule where the memory pointed to by the stack pointer is unmapped would be unsound. More precisely, whenever the stack memory is unmapped a push instruction will generate a cascade of exceptions that will cause the machine to reboot (Triple fault). The first exception is triggered by the push instruction writing to the top (unmapped) stack; the second exception is triggered by the interrupt mechanism trying to store the address of the last executed instruction on the top of the stack (Double Fault); and, similarly, the third exception will be generated by the interrupt mechanism trying to handle the second exception (Triple Fault).

The pop rule, similarly, reads from the stack and puts the value in the cell pointed to by the register r:

$$
\begin{array}{ll}
\vdash & \forall ij : \mathsf{DWORD}. \\
& (\triangleright(\mathsf{safe} \otimes (\mathsf{EIP} \mapsto \mathsf{fail} * \mathsf{INTL} \mapsto 1 * (sp-4)..sp \mapsto j * \mathsf{ESP} \mapsto (sp-4))) \\
& \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * \mathsf{INTL} \mapsto 0 * (sp-4)..sp \mapsto spv * \mathsf{ESP} \mapsto sp)) \\
& \quad \oslash(i..j \mapsto (\mathsf{pop}[r])) \\
& \quad \otimes \mathsf{IDT}[\mathsf{ExnGP}/\mathsf{fail}] * r \mapsto l * l..(l+4) \mapsto !!
\end{array}
\tag{11}
$$

the structure of the rule is again similar to the previous ones, and again the rule would be unsound if we tried to pop a value from a stack whose memory is unmapped.

5. Memory allocation using exceptions

In this section we show how to prove the memory allocator presented in Figure 2 correct. We first show how to reason about predicates that talk about chunks of memory with boundaries. More specifically, we need to be able to decide whether the current cell in the memory is available or not. Secondly, we use this result to prove the allocator correct.

5.1. The memory datatypes in Coq

As explained in Section 3.3, 32-bit binary addresses are represented by the DWORD type. On the other hand, the points-to relation is a function from Cursor 32 × Cursor 32 to an assertion logic formula such that for all p, q of type Cursor 32 and v of type DWORD, $p..q \mapsto v \vdash p \leq q$. The type Cursor 32 is making sure that p is actually smaller that q and that no wrap-around has occurred. When p and q are DWORDs there is a coercion into a Cursor 32.

For convenience, many instruction rules use syntactic sugar like $p \mapsto v$ as a notation for $\exists(q : \mathsf{Cursor}\ 32), p..q \mapsto v$. This is unfortunate as it implies that if we had more information about q by using these rules we would lose it. For example, assume we have a rule of the form $\{p \mapsto \_\}\ \mathsf{mov}[p], v\ \{p \mapsto v\}$ and that we want to prove $\{p..q \mapsto ?\}\ \mathsf{mov}[p], v\ \{p..q \mapsto v\}$. By applying the assumption we would have to prove that $\{p \mapsto v\}$ entails $\{p..q \mapsto v\}$, which is not provable, unless we extract some information about q before applying the assumption. In particular, there are two things we need to know. The first is that q was a DWORD and that $q = p + 4$, and the second is that q has not wrapped around. In order to retrieve these two pieces of information from $p..q \mapsto v$ we need to do some non-trivial Coq hacking.

We would like to point out that the only way to avoid this would have been to reformulate all the assembly rules with the explicit offset as above in the points-to relation. However, the problem arises again when the chunk of memory is represented by list. For example, assume that $p..q \mapsto s$ where s is a list of DWORDs. If we perform a case analysis on the list the first case is typically $s = a :: l$ for some a : DWORD and some list l. As a result we get that there

exists a cursor q' such that $p..q' \mapsto a$ and here we get into the same trouble again. Although this time we know q' is not top we still need a lemma in order to retrieve the piece of information that says that $q' = p + 4$. The lemma is stated as follows:

Lemma 1. *Let p, q be two* DWORD*s and v a value also of type* DWORD*, $p..q \mapsto v \vdash q = p + 4$*

This lemma it is somewhat easier than recovering that q has not wrapped around. However, it still required some non-trivial effort. The reader should note the subtlety of this lemma. In the above, p and q are cursors that came from a DWORD. Hence their value cannot be `top`. This means that the predicate $p..q \mapsto v$ is implicitly saying that $p + 4$ is not wrapping around since p and q are cursors and that in this datatype the addresses are sequential.

All in all, our hunch for a possible solution would be to define the points-to relation in such a way that it carries the information about what the second cursor is and to formulate a theory about cursors that allows us abandon the use of DWORDs.

By virtue of this effort we can now decide whether a chunk of memory is at the end of the memory address space or not:

Lemma 2. *Let base and limit be of type* Cursor 32 *and let buf be a list of memory cells of type* DWORD*. If $base..limit \mapsto buf * limit..(limit + 4) \mapsto !!$ then either*

$$\exists s_1.base..(base{+}4) \mapsto s_1 * \exists s_2.(base{+}4)..limit \mapsto s_2 * limit..(limit{+}4) \mapsto !! \quad (12)$$

or

$$base = limit \quad (13)$$

Proof. The proof is by case analysis on *buf*. If *buf* is the empty list then this implies *base* is equal to *limit*, thus satisfying (13).

If *buf* is composed of a cell a and a list l then there exists a p of type Cursor such that $base..p \mapsto a$ and $p..limit \mapsto l$. We proceed by case analysis on p. If p is a DWORD by Lemma 1 then p is equal $base + 4$, thus satisfying (12). If p is $\top$ then $base..\top \mapsto a$ is false. Since this was an assumption the case is vacuously true. □

5.2. Specification for the allocator

The specification for the example in Figure 2 has the following pattern,

$$\mathsf{allocSpec} \triangleq \vdash ((\mathsf{safe} \otimes Q_1 \wedge \mathsf{safe} \otimes Q_2) \Rightarrow \mathsf{safe} \otimes P) \oslash i..j \mapsto c \otimes \mathsf{Inv}$$

Two continuations, namely Q_1 and Q_2, are defined to state what happens upon success and failure, a pre-condition P together with an invariant Inv specifying that there exists a storage whose ends are bounded by an unmapped memory region and that there exists an IDT containing the pointers to the handlers.

More precisely, the precondition is defined as follows:

$$P \triangleq \mathsf{EIP} \mapsto i * \mathsf{INTL} \mapsto 0 * \mathsf{EDI}? * \mathsf{ESP} \mapsto sp * (sp{-}4..sp)? \tag{14}$$

The EIP points to the beginning of the code, INTL is the register keeping track of the level of interruptions, EDI is a temporary register and ESP is the stack pointer.

Upon failure the machine will jump to the handler, hence we have to ensure that the machine will be safe to run whenever this jump will be performed. This is expressed by the first of the two post-conditions:

$$Q_1 \triangleq \mathsf{EIP} \mapsto \mathsf{fail} * \mathsf{INTL} \mapsto 1 * \mathsf{EDI}? * \mathsf{ESP} \mapsto (sp{-}4) * (sp{-}4..sp)? \tag{15}$$

which states that there exists a handler which is going to take on the computation from the address $fail$ with the INTL set to 1 and the stack pointer containing the return address to the original code.

Also, we make sure the machine will be safe upon success. We do this by defining the other post-condition to be:

$$\begin{aligned} Q_2 \triangleq & \mathsf{EIP} \mapsto j * \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto sp * ((sp{-}4)..sp)? * \\ & \exists p, \mathsf{EDI} \mapsto (p{+}4) * (p..(p+4))? \end{aligned} \tag{16}$$

which states that there is a program which is safe to run from the address j with the EDI register pointing to the end of the allocated memory and with the interrupt level set at zero in case the allocator succeeds.

Furthermore, we have the following invariant:

$$\begin{aligned}\mathsf{Inv} \triangleq\ & \exists base\ count.infoBlock \mapsto base * \exists s, base..count \mapsto s \\ & * count..(count+4) \mapsto !! * \mathsf{IDTR} \mapsto idt \\ & * \mathsf{Flags} * \mathsf{IDT}[exn.ExnGP] \mapsto fail\end{aligned} \tag{17}$$

which states that the information block *infoBlock* points to a chunk of memory bounded at the top end by the unmapped region, and the Interrupt Descriptor Table is properly set up in the memory.

5.3. Proof of the specification

We prove the following theorem showing that implementation of the allocator respects the specification:

Theorem 1. *Let P, Q_1, Q_2 and* Inv *as respectively in (14), (15), (16) and (17). Moreover, let c be the code in Figure 2. The specification*

$$\vdash ((\mathsf{safe} \otimes Q_1 \wedge \mathsf{safe} \otimes Q_2) \Rightarrow \mathsf{safe} \otimes P) \oslash i..j \mapsto c \otimes \mathsf{Inv}$$

is sound. `[Coq proof]`

Proof. (Sketch). By unfolding the definition of the program there exists i_1, i_2, i_3, i_4, i_5 and i_6 of type DWORD pointing at each single instruction of the program:

$$\begin{aligned} i..i_1 &\mapsto \mathsf{mov\ ESI}, info * \\ i_1..i_2 &\mapsto \mathsf{mov\ EDI}, [\mathsf{ESI}] * \\ i_2..i_3 &\mapsto \mathsf{mov}\ [\mathsf{EDI}], 0 * \\ i_3..i_4 &\mapsto \mathsf{add\ EDI}, 4 * \\ i_4..i_5 &\mapsto \mathsf{mov}\ [\mathsf{EDI}], 0 * \\ i_5..i_6 &\mapsto \mathsf{add\ EDI}, 4 * \\ i_6..j &\mapsto \mathsf{mov}\ [\mathsf{ESI}], \mathsf{EDI} \end{aligned}$$

Proving the first two instructions correct is only a matter of applying the proper rules using the composition rule (5).

First, we apply the instruction rule for the first instruction by instantiating it with the proper parameters:

$$\begin{aligned}\vdash & \mathsf{safe} \otimes \mathsf{EIP} \mapsto i_1 * \mathsf{ESI} \mapsto info \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mapsto i * \mathsf{ESI}? \\ & \oslash i..i_1 \mapsto \mathsf{mov\ ESI}, info\end{aligned}$$

For the second instruction we apply the following rule:

$$\begin{aligned}\vdash & \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i_2 * \mathsf{EDI} \mapsto v * \mathsf{ESI} \mapsto info * info \mapsto base) \\ & \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i_1 * \mathsf{EDI}? * \mathsf{ESI} \mapsto info * info \mapsto base) \\ & \oslash i_1..i_2 \mapsto \mathsf{mov\ EDI}, [\mathsf{ESI}]\end{aligned}$$

We end up with the following precondition which we name P' with the program pointer pointing to the third instruction:

$$P' = \mathsf{safe} \otimes \mathsf{EIP} \mapsto i_2 * \mathsf{EDI} \mapsto \mathit{base} * \mathsf{ESI} \mapsto \mathit{info} * \mathit{info} \mapsto \mathit{base}$$
$$* \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto \mathit{sp} * (\mathit{sp} - 4)..\mathit{sp} \mapsto \mathit{spval} * \mathsf{Inv}$$

The instruction in i_2 is going to perform a write operation to the location pointed to by *base*. By unfolding the invariant Inv we know there exists *base* and *limit* of type DWORD bounding the memory:

$$\mathit{info} \mapsto \mathit{base} * \mathit{base}..\mathit{limit} \mapsto s * \mathit{limit}..(\mathit{limit} + 4) \mapsto !! * \mathsf{IDTR} \mapsto \mathit{idt} *$$
$$\mathsf{Flags} * \mathsf{IDT}[\mathsf{ExnGP}] \mapsto \mathit{fail}$$

on the other hand we do not know whether there is space left between them. So we perform case analysis on the memory chunk by applying Lemma 2, resulting in

$$\mathit{base}..(\mathit{base} + 4) \mapsto s * (\mathit{base} + 4)..\mathit{limit} \mapsto s * \mathit{limit}..(\mathit{limit} + 4) \mapsto !!$$
$$\vee\ \mathit{base} = \mathit{limit} * \mathit{base}..\mathit{base} + 4 \mapsto !!$$

This assertion is indeed part of P'. Thus, when applying (5) we will have to prove that P' implies the precondition of the instruction that we are going to use. This means the disjunction above will appear in the negative position. So we have to first split the disjunction into two sub cases. For the case in which the memory has run out we will apply (6) and for the other we will apply (3). In this case, we have that $\mathit{base} = \mathit{limit}$ and the exception is thrown. The move operation performs a write operation into the unmapped location. Let P'' be the precondition obtained from P' where $\mathit{base} = \mathit{limit}$. We use rule (6) instantiated as follows:

$$\triangleright(\mathsf{safe} \otimes (\mathsf{EIP} \mapsto \mathit{fail} * \mathsf{INTL} \mapsto 1 * \mathsf{ESP} \mapsto \mathit{sp} - 4 * (\mathit{sp} - 4)..\mathit{sp} \mapsto j))$$
$$\Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i2 * \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto \mathit{sp} * (\mathit{sp} - 4)..\mathit{sp} \mapsto \mathit{spval})$$
$$\oslash\ (i2..i3 \mapsto (\mathsf{mov}[\mathsf{EDI}], 0))$$
$$\otimes\ (\mathsf{EDI} \mapsto \mathit{limit} * \mathit{limit}..(\mathit{limit} + 4) \mapsto !! * \mathsf{IDTR} \mapsto \mathsf{IDT}[\mathsf{ExnGP}] \mapsto \mathit{fail})$$

This rule can be turned into a pattern suitable for applying (5) by commuting $\otimes$ with $\oslash$ and distributing $\otimes$ over the implication. Note that P'' entails the precondition of the instruction rule above and Q_2 entails the postcondition of the instruction rule. Hence the case is completed.

The second case is when the program goes through and it makes another write, after which there is a successful case and an unsuccessful one. These two cases are similar to the previous one, only that when the write is successful we have to prove the rest of the program correct, but this can be shown in the standard way. □

6. Related Work

The work most closely related to ours is naturally the work by Jensen et al. that we build on [14]. It allows specifications to be written in a clean and intuitive manner even for code that does not follow a basic-block like structure with only one entry and one exit point. It should be noted, however, that there are program logics that use Hoare triples on code with multiple exit points, such as programs containing break-statements from loops. One example is Appel's mechanised semantics library in Coq for C-minor [2] and the mechanisation in HOL4 of x86 and ARM assembly by Myreen et al. [16]. Both of these mechanisations have special postconditions that are used to handle unstructured control flow. Other relevant work includes Chlipala's Bedrock framework that allows assembly-like programs to be verified in Coq [10]. Chlipala's work focusses heavily on automation. Its specification logic (XCAP) is an higher-order separation logic which allows to prove properties about programs that pass around other programs using code pointers. On the other hand, the logic and semantics do not support code as data.

Our work has similarities with Crash Hoare Logic (CHL) [9]. In modern operating systems, whenever a crash or a reboot occurs, a filesystem is able to restore the data to a consistent state. CHL is a Hoare logic to prove correct filesystems with recovery facilities of this kind. The specification logic has pre and postconditions as in standard Hoare logic with an additional crash condition which states that if a crash occurred the data stored on the disk is consistent.

We adopt a very similar idea in that we add an additional condition in case an exception occurred. The advantage of having an unstructured specification logic is that we can express the crash condition as an assertion composed with the original postcondition by using the logical operator $\wedge$ (and).

None of the work mentioned above currently support interrupts. Seminal work on mechanising interrupts was made by Feng et al. [12]. They formalise what they call an Abstract Interrupt Machine (AIM) in two layers. The first is a machine in which programs are interrupt-aware, i.e. they have access to interrupt

low-level instructions (e.g. sti, cli and iret). The second defines an extension of the first by adding thread queues and extending the language with explicit primitives for manipulating the thread queue. For the logic they employ a rely-guarantee separation logic with preconditions only. The handler is specified in such a way that the memory of the thread is guaranteed to be left untouched. More recent work [8] focusses on certifying device drivers that use the interrupt mechanism to communicate with the operating system.

On the other hand, our work focusses on handcrafted unstructured low-level code whose semantics allow for self-modification. This presents additional challenges as it prevents us to have more abstractions (unless we implement and verify them on top of the existing formalisation) and automation to rely on, but on the other hand, allows us to verify a wider class of programs.

7. Conclusions and Future Work

This is preliminary work towards formalising the interrupt mechanism which was necessary to understand the potential of the proposed logic and ultimately move towards the formalisation of timer handlers and schedulers in uniprocessor machines like the Intel x86.

As a first step towards this goal, we have extended an existing mechanisation of x86-assembly to support synchronous interrupts. By using Jensen's model we inherit a fairly concrete model of the Intel x86 machine to reason about mutable code and we stay true to this design philosophy by storing the IDT and all handlers in memory, allowing them to be dynamically updated by the processor. Our extensions to the program logic are also very conservative. By allowing the memory points-to predicate to state that certain memory is unmapped (and not only what it contains), we obtain a logic that is expressive enough to verify programs that use synchronous interrupts. In particular, our example proves that the logic is suitable for formalising programs which generate a general protection exception. By conservatively turning all errors into exceptions we guarantee that sound rules in the old formalisation remain sound and that new

rules can be added when needed to handle other kinds of exceptions with the same pattern we used.

We believe that this is a testament not only to the validity of our design decisions, but also of the quality of the original mechanisation.

As future work, we would like to formalise asynchronous interrupts, thus ultimately being able to verify timer handlers, schedulers and concurrency. However, when an interrupt fires at an unpredictable point in time it introduces interference. This is true even for interrupts that do not share memory with other threads. In fact, flags and registers in the model are treated as part of the memory, and as a result, so is the stack. Indeed, there are many points of inspiration from Feng et al.'s work, such as rely and guarantee separation logic to ensure well-behaved handlers restore the stack and the flags correctly. A more modular way of reasoning about shared mutable state would be that of Svendsen and Birkedal's work [21]. Indeed, it would be interesting to investigate the connections of both logics with the tensor operator ($\otimes$).

Finally, in this paper we proved the correctness of only one kind of exception, which we deemed of most interest. We think verifying exceptions will be much more useful once we assume concurrency – or implement it by means of a verified scheduler as in Feng et al [12]. In particular, we could use our machinery to verify that even if a thread throws an exception the machine continues running. In particular, the process is killed by the operating system which then schedules another one.

8. Acknowledgements

We would like to thank Jonas Jensen for fruitful discussions and the anonymous reviewers for useful comments on improving the motivational example and the overall quality of the paper.

[1] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007.

[2] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[3] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! – a framework for higher-order separation logic in Coq. In *ITP*, 2012.

[4] Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. In *ITP*, 2011.

[5] L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 2012.

[6] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.

[7] Cristiano Calcagno, Peter W. O'Hearn, and Richard Bornat. Program logic and equivalence in the presence of garbage collection. *Theor. Comput. Sci.*, 2003.

[8] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. In *PLDI*, 2016.

[9] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, 2016.

[10] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, 2011.

[11] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B & 2C): Instruction Set Reference*, January 2013.

[12] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, 2008.

[13] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[14] J. B. Jensen, N. Benton, and A. J. Kennedy. High-level separation logic for low-level code. In *POPL*, 2013.

[15] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. Coq: the world's best macro assembler? In *PPDP*, 2013.

[16] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In *TACAS*, 2007.

[17] H. Nakano. A modality for recursion. In *LICS*, 2000.

[18] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[19] Uday S. Reddy and Hongseok Yang. Correctness of data representations involving heap data structures. *Sci. Comput. Program.*, 2004.

[20] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[21] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.

[22] The Coq Development Team. *The Coq Reference Manual, version 8.4*, 2012.

Appendix A. Lemmas proved in Coq

This section is for review purposes only. Here, we report the contributions we made to the Coq development.

The main example is contained in the Coq file `allocexp.v`: `alloccases` is an auxiliary lemma, given a memory region either the cell pointed to is present

in the memory or is unmapped; `allocSpec` is the specification of the allocator; `allocInv` is the invariant that contains the IDT and the memory region.

```
----------------------
./coqdev/x86/allocexp.v
----------------------
Lemma alloccases
Lemma inlineAlloc_correct
Definition allocSpec
Definition allocImp
Definition allocInv
```

In `cursor.v` we added a couple of lemmas for reasoning about cursors.

```
----------------------
./coqdev/cursor.v
----------------------

Lemma inRange_tau
Lemma inRange_tau2
```

In `pointsto.v` we define the points-to relation for the unmapped memory. Points-to relations for the memory are instances of the Type Class `MemIs` previously defined in the same file.

```
----------------------
./coqdev/pointsto.v
----------------------
Program Instance memIsUnmapped: MemIs unmap
Notation "!!" := unMap.
Corollary memIsDWORD_range
Lemma memIsDWORD_no_wraparound_aux
Lemma memIsDWORD_no_wraparound
Lemma memIsDWORD_q_distance
```

This Coq file `triple.v` contains old-style Hoare Triples for reasoning *inside* the instructions, in other words, to reason about micro-instructions. Since we add a catch function to the semantics we need new triples to deal with it.

```
----------------------
./coqdev/triple.v
----------------------
triple_memIs_setDWORDSep
```

```
memIsDWORD_q_distance
memIsDWORD_range

Lemma triple_catchLetGetReg
Lemma triple_catchLetGetFlag
Lemma triple_catchSetRegSep
Lemma triple_catchLetGetRegSep
Lemma triple_catchLetGetSepGen
Lemma triple_catchLetGetSep
Lemma triple_catchLetGetDWORDSep
Lemma triple_catchLetGetDWORDSepGen
Lemma triple_catchLetGetBYTESep

Lemma elim_catch: forall P Q c,
  TRIPLE P c Q -> TRIPLE P (catch c) Q .

Corollary readMem_unmapped_BYTE
Corollary readMem_unmapped_DWORD
Corollary readMem_unmapped_DWORD_or_BYTE
Lemma setDWORD_unmapped
Lemma triple_catchRaiseDWORD
Lemma triple_catchRaiseDWORD_2
Lemma byteIsNotMapped
Lemma retrieve_distanceGen
Lemma retrieve_distance
Lemma triple_catchRaise
Lemma triple_raise
Lemma triple_catchDoSetDWORDInProcStateExp
Lemma triple_catchLetGetDWORDInProcStateExp
Lemma triple_catchLetGetDWORDorBYTEInProcStateExp
Lemma triple_catchSetDWORDInProcStateExp

triple_memIs_setDWORDSep
triple_memIs_setDWORDSep
```

The Coq file `instrrules.v` contains the instruction rules we added in the specification logic. We modified some of the instructions that use the specification $i..j \mapsto v$ instead of $p \mapsto v$.

```
----------------------
./coqdev/x86/instrrules.v
----------------------
Lemma evalReg_rule_catch
Lemma getReg_rule_catch
Lemma evalMemSpec_rule_catch
```

```
Lemma PUSH_M0_rule_exp_src
Lemma MOV_MC_rule_exp
Lemma MOV_RMb_rule_exp
Lemma MOV_MV_rule_memIs
```