Automated Verification of a Component Platform

Matthew Fernandez, June Andronick, Gerwin Klein, Ihor Kuz

August 2015

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. NICTA is also funded and supported by the Australian Capital Territory, the New South Wales, Queensland and Victorian Governments, the Australian National University, the University of New South Wales, the University of Melbourne, the University of Queensland, the University of Sydney, Griffith University, Queensland University of Technology, Monash University and other university partners.

Copyright © 2015 NICTA, ABN 62 102 206 173. All rights reserved except those specified herein.

This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S.Government.

Abstract

This document outlines various criteria defining the correctness of a component platform, and describes a process for automatic verification of some of these criteria for the CAmkES component platform. Generated proofs are provided for a sample system as an example of the output artefacts and their relationship to both hand written formal reasoning and generated code. Our correctness properties depend upon an interactive theorem prover and some straightforward translation tools. We consider our assumptions and resulting trusted computing base to be acceptable in a high assurance software environment.

This report is an extension to previous reports on the formalisation of the CAmkES component platform and a familiarity with these previous documents is assumed.

Contents

1. Introduction

When building a high assurance software system, the correctness of the system is crucial to its functionality. For large systems, this can be challenging, and component-based design is a common technique used to manage the engineering complexity in such systems. However, in discussions of the correctness of component-based systems, an assumption that is often implicit is the correctness of the component platform itself. The component platform implements the isolation between components and the communication primitives the components depend upon, and thus bugs in the platform can undermine and invalidate any guarantees of the composition on top. A more critical view of the correctness of a component-based system encompasses not just the components, but the component platform and even the underlying operating system.

Taking up this challenge, we start with a formally verified, general purpose microkernel, seL4 [6]. To build component-based systems, we use the CAmkES component platform [7], a static platform optimised for seL4. Our aim is to take verification to the next level: to leverage the properties of seL4 to provide a high assurance environment for component-based development. Our vision involves generated proofs for generated code, leaving only hand written component code requiring manual effort.

This report describes how we have automated correctness proofs for the generated code implementing CAmkES' Remote Procedure Call (RPC) communication. Guarantees about generated code are provided in the form of proofs in the interactive theorem prover, Isabelle/HOL [8], and build on prior tools for reasoning about C code [5, 9]. We assume the correctness of the theorem prover itself and the correctness of an initial translation from C code into SIMPL, but we do not assume the correctness of other transformations inside the theorem prover. In particular, further code abstraction on SIMPL representations of C code that is performed inside Isabelle/HOL has been proven to preserve the semantics of the original representation.

This report extends previous reports on the static [4] and dynamic [3] semantics of CAmkES and the safety of RPC glue code [1]. Where the preceding report showed safe execution of the RPC glue code, the present report extends this to functional correctness. This enables hand written proofs to be manually composed with generated proofs to form an overall guarantee of the functional correctness of a component-based system.

The guarantees we provide for component-based systems will be introduced with reference to an example system that is presented in Chapter 3. The exact generated theorems for this example system are detailed in Chapter 5, with their unabbreviated proofs provided in Appendix B. We show the expected work flow for composing these proofs with hand written proofs in Chapter 6. The benefits provided by this work and planned extensions are summarised in Chapter 7. Parts of the work contained in this report have been the subject of a recent paper [2]. Interested readers are advised to refer to this publication for further, complementary information.

2. Defining Correctness

Correctness of a componentised system is a property of the entire system. That is to say, system correctness is the conjunction of the correctness of each constituent part of a componentised system. In a static component platform such as CAmkES, correctness can be decomposed into correctness of the components themselves, correctness of CAmkES and correctness of seL4. Correctness of the components themselves is a system-specific property and cannot be defined generically in a useful form. The correctness of seL4 has been the subject of prior work and is not dealt with in this report. The following chapters of this report are concerned exclusively with the correctness of CAmkES, the component platform, itself.

"Correctness" is an inherently ambiguous term, for which there are many potential candidate properties that could be formalised. The property we choose to focus on is the correctness of the generated communication code for RPC communication. More precisely, the desired property is that a remote function invocation (via a CAmkES-provided RPC mechanism) is equivalent to a local invocation of the same function. We choose this property as indicative of correctness of communication in the CAmkES component platform, because it is a complex claim and one that is not observably true a priori from manual inspection of generated code. We elaborate on this motivation in prior published work [2]. This is the claim of the final theorem we will introduce in Chapter 5.

With this property available, a user is liberated to reason about their component-based system as if RPC communication was simply a local function call. Properties that they derive under this abstraction of RPC invocations as function calls remain valid when introducing the complete semantics of CAmkES primitives, as introduced assumptions are discharged by the generated theorems the platform supplies. By composing these generated and hand written proofs together, with the pre-existing kernel correctness guarantees, it is possible to achieve a whole system assurance property while only manually reasoning about the hand written component code.

3. Example System

This section describes a simple example system that will be referred to in following sections to demonstrate how our verification techniques would be applied to a real system. Th example is intentionally minimal to be as comprehensible as possible, but the techniques we apply generalise to arbitrary CAmkES systems.

CAmkES systems are typically conceived as architectural diagrams of boxes representing component instances and lines between them representing connections. Our example system consists of two component instances, `s` and `c`, that communicate over a connection, `conn` (depicted in Figure 3.1).

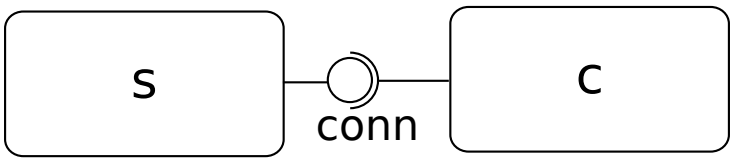

Figure 3.1.: Swap counter system

While Figure 3.1 is useful as an abstraction when thinking about the system, the precise architectural specification of the system is given in Figure 3.2. This describes two component types, `Client` and `Service`, each of which is instantiated once (as `c` and `s`, respectively). An RPC interface is defined, `Swapper`, which `Client` uses and `Service` provides. The effect of this is that `s` implements this procedure under the name `ss` and `c` expects to be able to call this procedure under the name `cs`. The details of the procedure implementation itself will be described later.

```
1  procedure Swapper {
2    unsigned int swap(inout int a, inout int b);
3  }
4
5  component Client {
6    control;
7    uses Swapper cs;
8  }
9
10 component Service {
11   provides Swapper ss;
12 }
13
14 assembly {
15   composition {
16     component Client c;
17     component Service s;
18     connection seL4RPCSimple conn(from c.cs, to s.ss);
19   }
20 }
```

Figure 3.2.: Swap counter specification

The connection between the two component instances, `conn`, has a type, `seL4RPCSimple`. This type is

attached to a code template that determines the mechanism for communication.

Both component types require some code to implement their behaviour. `Client` is an active component, indicated by the `control` keyword, and hence it needs a `run` function as an entry point, shown in Figure 3.3.

```
1  int run(void) {
2      int x = 3;
3      int y = 5;
4      unsigned int i;
5      i = cs_swap(&x, &y);
6      return 0;
7  }
```

Figure 3.3.: Client C code

`Service` implements the `Swapper` procedure and its code is shown in Figure 3.4.

```
1  static unsigned int counter;
2
3  unsigned int
4  ss_swap(int *a, int *b){
5      int temp = *a;
6      *a = *b; *b = temp;
7      counter++;
8      return counter;
9  }
```

Figure 3.4.: Service C code

Already we can see the intended functionality of this example system; namely, that `Service` provides a way to swap two integer pointers and `Client` uses this functionality to swap two of its own pointers. Note that the code is written as if `cs_swap` and `ss_swap` referred to the same function. That is, as if a function call to `cs_swap` resulted in execution of `ss_swap`. This is the precise abstraction view provided by RPC in component platforms and the correctness of this abstraction is the goal of our generated proofs.

4. Syntax

The syntax we use for expressing properties about code was introduced in a previous report [1], but we re-introduce it here for clarity. To express properties of code, we use similar notation to Hoare logic. The following claims that, given a pre-condition `P`, a post-condition `Q` holds after executing the function `f`.

```
"⦇P⦈ f ⦇Q⦈"
```

As it stands, this specification admits a function that never terminates. We use a "!" to express termination. So an equivalent specification that also claims that `f` does not fail is the following.

```
"⦇P⦈ f ⦇Q⦈!"
```

Our existing C toolchain [5, 9] allows us to prove properties over specifications like the above.

The generated theorems that we will introduce in Chapter 5 are produced inside a `locale`. A `locale` is a way of capturing a collection of parameters and assumptions about the contained formalism. In our case, these assumptions are properties of hand written code that we expect a user to provide when composing the generated proofs with their own hand written proofs. This process of instantiating a `locale` with specific parameters and discharging the associated assumptions is referred to as `interpretation`.

5. Generated Theorems

This section will present RPC correctness theorems defined on a representation of C code produced by the SIMPL-to-C parser and AutoCorres. The definitions and proofs are introduced in a certain order to aid comprehension, but it is important to emphasise that both the code that is the subject of the proofs and the proofs themselves are generated automatically by CAmkES. The proofs are elided in this section, but the full proofs are provided in Appendix B.

In order to talk about the correctness of marshalling and unmarshalling operations, we need a way to describe the contents of the Inter-Process Communication (IPC) buffer. To achieve this, we introduce a predicate, `packed`, that takes a list of 32-bit words and asserts that the first `n` words of the IPC buffer have these values, where `n` is the length of the list.

definition

```
packed :: "lifted_globals ⇒ 32 word list ⇒ bool"
```

We can now use this predicate to describe the correctness of the caller's (`c`'s) marshalling code. The following lemma claims that, from a valid initial state, after executing `c`'s marshalling code, the IPC buffer contains the call arguments.

lemma

```
"∀s'. {|λs. inv s ∧ s' = setMR (setMR (setMR s 0 0) 1 (scast p0)) 2 (scast p1)|}
        cs_swap_marshal p0 p1
      {|λr s. s = s' ∧ inv s ∧ r = 3 ∧ packed s [0, scast p0, scast p1]|}!"
```

The proof of this lemma has been elided for conciseness, but it is generated alongside the lemma itself.

We can then take this state, with the call arguments in the IPC buffer, and state the following lemma for correctness of the callee's (`s`'s) unmarshalling code. This lemma claims that, after unmarshalling, `s`'s pointers `p0` and `p1` contain the call arguments. Note that we need extra pre-conditions requiring that `p0` and `p1` are valid pointers and are not the same pointer.

lemma

```
"∀s₀ p0' p1'. {|λs. s = s₀ ∧ inv s ∧ ptr_valid_s32 s p0 ∧ ptr_valid_s32 s p1 ∧
                    distinct [p0, p1] ∧ packed s [0, p0', p1']|}
                ss_swap_unmarshal p0 p1
              {|λ_ s. inv s ∧ ptr_contains_s32 s p0 (ucast p0') ∧
                      ptr_contains_s32 s p1 (ucast p1') ∧
                      s = update_s32 (update_s32 s₀ p0 (ucast p0')) p1 (ucast p1')|}!"
```

The lemmas we have seen thus far capture the required correctness properties of the RPC communication involved in making the call. It remains to do the same for the communication involved in the return from the call. First we provide the equivalent marshalling lemma for the callee, `s`. This expresses the necessary correctness property for marshalling of the return values of the RPC invocation.

lemma

```
"∀s₀. {|λs. s = s₀ ∧ inv s|}
```

```
    ss_swap_marshal ret p0 p1
  {|λr s. inv s ∧ r = 3 ∧
        s = setMR (setMR (setMR s₀ 0 ret) 1 (scast p0)) 2 (scast p1) ∧
        packed s [ret, scast p0, scast p1]|}!"
```

We now need a lemma for the correctness of unmarshalling the return values by the caller, `c`. The following lemma captures this, and note once again that we require the pointers we are unmarshalling return values into to be both valid and distinct.

lemma

```
"∀s₀ ret p0' p1'.
  {|λs. s = s₀ ∧ inv s ∧ ptr_valid_s32 s p0 ∧ ptr_valid_s32 s p1 ∧
       distinct [p0, p1] ∧ packed s [ret, p0', p1']|}
    cs_swap_unmarshal p0 p1
  {|λr s. inv s ∧ r = ret ∧ ptr_contains_s32 s p0 (ucast p0') ∧
         ptr_contains_s32 s p1 (ucast p1') ∧
         s = update_s32 (update_s32 s₀ p0 (ucast p0')) p1 (ucast p1')|}!"
```

Finally, with these constituent pieces, we can form a correctness lemma for the RPC glue code as a whole. The following lemma states that, given certain requirements of user pointers previously discussed, we can transfer a user's pre- and post-condition from their implementation to a remote invocation of their implementation. Note that we accrue an extra set of pre-conditions that describe the invariance of the user's post-condition on the addresses of the pointers to unmarshal return values into. While the user's post-condition is expected to depend on the *values* at these addresses, it is not expected to depend on the addresses themselves.

lemma

```
"∀s₀. {|λs. s = s₀ ∧ inv s ∧ ptr_valid_s32 s p0' ∧ ptr_valid_s32 s p1' ∧
           distinct [p0', p1'] ∧
           (∀s₁ s₂ v. swap_Q s₁ (update_s32 s₂ p0' v) = swap_Q s₁ s₂) ∧
           (∀s₁ s₂ v. swap_Q s₁ (update_s32 s₂ p1' v) = swap_Q s₁ s₂) ∧
           swap_P s p0 p1|}
        do cs_swap_marshal p0 p1;
           ss_swap_internal;
           cs_swap_unmarshal p0' p1'
        od
      {|λr s. inv s ∧
            swap_Q s₀ s r p0 (ptr_value_s32 s p0') p1 (ptr_value_s32 s p1')|}!"
```

This final lemma can be utilised in further hand written proofs about correctness of hand written code that calls the RPC glue code.

6. Composition

Taking the generated theorems from Chapter 5, we now show how to compose these with hand written proofs to form a whole system guarantee. First, let us consider the correctness property of the `ss_swap` function that was introduced in Figure 3.4. The intuition is that this function is correct if, following its execution, the values to which the two pointers refer have been swapped, the counter has been incremented and no other memory has changed. Considering only the *values* of the inputs, not their *addresses*, the pre-condition of this function is trivial.

definition
```
  P :: "lifted_globals ⇒ 32 signed word ⇒ 32 signed word ⇒ bool"
```
where
```
  "P s₀ x y ≡ True"
```

For the post-condition, given initial state s_0, final state `s`, return value `r`, input values `x` and `y` and output values `x'` and `y'`, we can pose a definition which expresses that the values have been swapped and the counter incremented.

definition
```
  Q :: "lifted_globals ⇒ lifted_globals ⇒ 32 word ⇒ 32 signed word ⇒
          32 signed word ⇒ 32 signed word ⇒ 32 signed word ⇒ bool"
```
where
```
  "Q s₀ s r x y x' y' ≡ r = counter s₀ + 1 ∧ counter s = r ∧
                         is_valid_w32 s₀ = is_valid_w32 s ∧
                         y' = x ∧ x' = y"
```

We can now state the correctness of the function implementation as a Hoare triple. Note that in this lemma we need to refer to dereferenced pointers and separately claim the validity of the pointers in the pre-condition. This form may seem counter-intuitive, but it becomes useful when specialising the generated proofs to this pre- and post-condition we have just stated.

lemma
```
  "∀s₀. {|λs. s = s₀ ∧ ptr_valid_s32 s x ∧ ptr_valid_s32 s y ∧
              P s (ptr_value_s32 s x) (ptr_value_s32 s y) ∧ inv s|}
        ss_swap x y
      {|λr s. Q s₀ s r (ptr_value_s32 s₀ x) (ptr_value_s32 s₀ y)
                (ptr_value_s32 s x) (ptr_value_s32 s y) ∧ inv s|}!"
```

The proof of this lemma is trivial and has been elided for conciseness.

We are now in a position to specialise the generated proofs with our pre- and post-condition. To do this, we use the `interpretation` command and provide the pre- and post-conditions we previously defined.

interpretation `conn_glue'  P Q`

7. Conclusion

This report documents the automated theorems and proofs currently provided by the CAmkES component platform, and their application to a simple example. While the example shown contains only a single procedure, the automation generalises to an arbitrary number of procedures and scales linearly with the number of RPC functions. The connector type we have discussed in this report, `seL4RPCSimple`, differs from the connectors currently in use in our research vehicle prototype, but we intend to apply the same verification techniques to the other connectors in use in the future. We have shown how to manually compose generated theorems with hand written proofs, and we intend to provide more tools to support this manual step in future. With the process we have shown, it is possible to achieve strong assurance in a component-based system, with a small set of trusted assumptions. By applying the techniques from this report, the costs associated with formal verification can be minimised and users can focus on reasoning about the correctness of their own hand written code.

Bibliography

[1] Matthew Fernandez, June Andronick, Gerwin Klein, and Ihor Kuz. CAmkES glue code proofs. Technical report, NICTA and UNSW, Australia, May 2014.

[2] Matthew Fernandez, June Andronick, Gerwin Klein, and Ihor Kuz. Automated verification of RPC stub code. In *International Symposium on Formal Methods*, pages 273–290, Oslo, Norway, June 2015.

[3] Matthew Fernandez, Peter Gammie, June Andronick, Gerwin Klein, and Ihor Kuz. CAmkES glue code semantics. Technical report, NICTA and UNSW, Australia, November 2013.

[4] Matthew Fernandez, Gerwin Klein, Ihor Kuz, and Toby Murray. CAmkES formalisation of a component platform. Technical report, NICTA and UNSW, Australia, November 2013.

[5] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. pages 429–439, Edinburgh, UK, June 2014.

[6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. pages 207–220, Big Sky, MT, USA, October 2009.

[7] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5):687–699, May 2007.

[8] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[9] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge Computer Laboratory, 1998.

Appendices

A. Generated Code

This appendix contains the generated RPC code for component instances in the example system introduced in Chapter 3. We first provide the generated code for `c`.

```
1  int cs__run(void) {
2    /* Nothing to be done. */
3    return 0;
4  }
5
6  static unsigned int cs_swap_marshal(int a, int b) {
7    unsigned int _camkes_length_8 = 0;
8    /* Marshal the method index. */
9    seL4_SetMR(_camkes_length_8, 0);
10   _camkes_length_8++;
11   seL4_SetMR(_camkes_length_8, (seL4_Word)a);
12   _camkes_length_8++;
13   seL4_SetMR(_camkes_length_8, (seL4_Word)b);
14   _camkes_length_8++;
15   return _camkes_length_8;
16 }
17
18 static void cs_swap_call(unsigned int length) {
19   /* Call the endpoint */
20   seL4_MessageInfo_t info = seL4_MessageInfo_new(0, 0, 0, length);
21   (void)seL4_Call(3, info);
22 }
23
24 static unsigned int cs_swap_unmarshal(int *a, int *b) {
25   unsigned int _camkes_mr_9 = 0;
26   unsigned int _camkes_ret_10 = (unsigned int)seL4_GetMR(_camkes_mr_9);
27   _camkes_mr_9++;
28   *a = (int)seL4_GetMR(_camkes_mr_9);
29   _camkes_mr_9++;
30   *b = (int)seL4_GetMR(_camkes_mr_9);
31   _camkes_mr_9++;
32   return _camkes_ret_10;
33 }
34
35 unsigned int cs_swap(int *a, int *b) {
36   /* Marshal input parameters. */
37   unsigned int _camkes_mr_index_11 = cs_swap_marshal(*a, *b);
38   /* Call the endpoint */
39   cs_swap_call(_camkes_mr_index_11);
40   /* Unmarshal the response */
41   unsigned int _camkes_ret_12 = cs_swap_unmarshal(a, b);
42   return _camkes_ret_12;
43 }
```

Below we provide the generated code for `s`.

```
1  extern unsigned int ss_swap(int *a, int *b);
2
3  static int swap_a_1;
4
5  static int swap_a_2;
6
7  static int *get_swap_a(void) __attribute__((__unused__));
8
```

```
 9  static int *get_swap_a(void) {
10    switch (camkes_get_tls()->thread_index) {
11    case 1:
12      return &swap_a_1;
13    case 2:
14      return &swap_a_2;
15    default:
16      (void)0;
17      abort();
18    }
19  }
20
21  static int swap_b_1;
22
23  static int swap_b_2;
24
25  static int *get_swap_b(void) __attribute__((__unused__));
26
27  static int *get_swap_b(void) {
28    switch (camkes_get_tls()->thread_index) {
29    case 1:
30      return &swap_b_1;
31    case 2:
32      return &swap_b_2;
33    default:
34      (void)0;
35      abort();
36    }
37  }
38
39  static void ss_swap_unmarshal(int *a, int *b) {
40    unsigned int _camkes_mr_13 = 1;
41    /* 0 contained the method index. */
42    *a = seL4_GetMR(_camkes_mr_13);
43    _camkes_mr_13++;
44    *b = seL4_GetMR(_camkes_mr_13);
45    _camkes_mr_13++;
46  }
47
48  static unsigned int ss_swap_invoke(int *a, int *b) {
49    /* Call the implementation */
50    return ss_swap(a, b);
51  }
52
53  static unsigned int ss_swap_marshal(unsigned int _camkes_ret_14, int a, int b) {
54    unsigned int _camkes_mr_15 = 0;
55    seL4_SetMR(_camkes_mr_15, (seL4_Word)_camkes_ret_14);
56    _camkes_mr_15++;
57    seL4_SetMR(_camkes_mr_15, (seL4_Word)a);
58    _camkes_mr_15++;
59    seL4_SetMR(_camkes_mr_15, (seL4_Word)b);
60    _camkes_mr_15++;
61    return _camkes_mr_15;
62  }
63
64  static unsigned int ss_swap_internal(void) {
65    int *a = get_swap_a();
66    int *b = get_swap_b();
67    ss_swap_unmarshal(a, b);
68    unsigned int _camkes_ret_16 = ss_swap_invoke(a, b);
69    unsigned int _camkes_length_17 = ss_swap_marshal(_camkes_ret_16, *a, *b);
70    return _camkes_length_17;
71  }
72
73  static seL4_MessageInfo_t ss__run_internal(_Bool _camkes_first_19,
74                                             seL4_MessageInfo_t _camkes_info_18) {
75    if (_camkes_first_19) {
```

```
76      _camkes_info_18 = seL4_Wait(3, ((void *)0));
77    }
78    /* We should have at least been passed a method index */
79    (void)0;
80    int _camkes_call_20 = (int)seL4_GetMR(0);
81    switch (_camkes_call_20) {
82    case 0: {
83      /* swap */
84      unsigned int _camkes_length_21 = ss_swap_internal();
85      /* Send the response */
86      _camkes_info_18 = seL4_MessageInfo_new(0, 0, 0, _camkes_length_21);
87      seL4_ReplyWait(3, _camkes_info_18, ((void *)0));
88      break;
89    }
90    default: { (void)0; }
91    }
92    return _camkes_info_18;
93  }
94
95  int ss__run(void) {
96    _Bool _camkes_first_22 = 1;
97    seL4_MessageInfo_t _camkes_info_23 = {{0}};
98    while (1) {
99      _camkes_info_23 = ss__run_internal(_camkes_first_22, _camkes_info_23);
100     _camkes_first_22 = 0;
101   }
102   return 0;
103 }
```

B. Generated Proofs

This appendix contains the entirety of the generated proofs for the example system introduced in Chapter 3.

theory conn **imports**
  "~~/../l4v/tools/autocorres/AutoCorres"
  "~~/../l4v/lib/LemmaBucket"
  "~~/../l4v/lib/WordBitwiseSigned"
begin

The following are various generic supporting lemmas that are emitted as static proof text.

lemma circ_break:"⟦f = g; h = i⟧ ⟹ f ∘ h = g ∘ i"
  by fast

lemma update_fn_unwrap:
  "Q = Q' ⟹ (λa b. if P a b then Q a b else R a b) = (λa b. if P a b then Q' a b else
R a b)"
  by clarsimp

lemma chunk_64[simp]:
  "ucast (
    ((ucast ((scast (x::64 sword))::32 word))::64 word) ||
    (((ucast ((ucast (((scast x)::64 word) >> 32))::32 word))::64 word) << 32)
) = x"
  by word_bitwise_signed

lemma chunk_s64[simp]:
  "scast (
    ((ucast (x::32 word))::64 sword) || ((ucast (((ucast (y::32 word))::64 word) << 32))::64
sword)) =
    ((ucast x)::64 word) || (((ucast y)::64 word) << 32)"
  by word_bitwise_signed

lemma validNF_intro_binder:"{|P|} f {|Q|}! ⟹ ∀s. {|P|} f {|Q|}!"
  by simp

declare [[allow_underscore_idents=true]]

We import and abstract the combined C sources of the generated code.

install_C_file "document/conn.c"

autocorres [ts_rules = nondet, skip_word_abs] "document/conn.c"

Various properties and definitions of the imported code have been stored inside a `locale`, which we now enter and extend with some further elements.

context conn **begin**

The definition of the function `abort` is not provided in the imported sources. Instead of giving it a complete specification, we provide the following that claims this function always fails. The effect of this is an introduced obligation to prove that any code that calls `abort` is unreachable.

```
lemma abort_wp[wp]:
  "⦇bottom⦈ abort' ⦇P⦈!"
  by (rule validNF_false_pre)
```

The following definition expresses a pointer to the current thread's IPC buffer.

```
definition
  ipc_buffer_ptr :: "lifted_globals ⇒ seL4_IPCBuffer__C ptr"
where
  "ipc_buffer_ptr s ≡ heap_seL4_IPCBuffer__C'ptr s (Ptr (scast seL4_GlobalsFrame))"
```

Similarly, the following defines the IPC buffer itself.

```
definition
  ipc_buffer :: "lifted_globals ⇒ seL4_IPCBuffer__C"
where
  "ipc_buffer s ≡ heap_seL4_IPCBuffer__C s (ipc_buffer_ptr s)"
```

We introduce a predicate to capture that the globals frame (a region of memory where seL4 stores a pointer to the IPC buffer) has not been modified by the application.

```
definition
  globals_frame_intact :: "lifted_globals ⇒ bool"
where
  "globals_frame_intact s ≡
    is_valid_seL4_IPCBuffer__C'ptr s (Ptr (scast seL4_GlobalsFrame))"
```

The second significant part of a global invariant we need is a predicate that the user has a valid IPC buffer.

```
definition
  ipc_buffer_valid :: "lifted_globals ⇒ bool"
where
  "ipc_buffer_valid s ≡ is_valid_seL4_IPCBuffer__C s (ipc_buffer_ptr s)"
```

The following two definitions capture a pointer to the Thread-Local Storage (TLS) region and the TLS region itself.

```
definition
  tls_ptr :: "lifted_globals ⇒ camkes_tls_t_C ptr"
where
  "tls_ptr s ≡ Ptr (ptr_val (heap_seL4_IPCBuffer__C'ptr s
     (Ptr (scast seL4_GlobalsFrame))) && 0xFFFFF000)"

definition
  tls :: "lifted_globals ⇒ camkes_tls_t_C"
where
  "tls s ≡ heap_camkes_tls_t_C s (tls_ptr s)"
```

The third part of the global invariant we require is that the TLS region is valid.

```
definition
  tls_valid :: "lifted_globals ⇒ bool"
where
  "tls_valid s ≡ is_valid_camkes_tls_t_C s (tls_ptr s)"
```

The number of threads in the current system. Note that CAmkES has this information at the time of code generation.

definition
```
  thread_count :: "32 word"
```
where
```
  "thread_count ≡ 2"
```

definition
```
  thread_index :: "lifted_globals ⇒ 32 word"
```
where
```
  "thread_index s ≡ thread_index_C (tls s)"
```

definition
```
  thread_valid :: "lifted_globals ⇒ bool"
```
where
```
  "thread_valid s ≡ thread_index s ∈ {1..thread_count}"
```

We are now at a point where we can introduce the global invariant itself.

definition
```
  inv :: "lifted_globals ⇒ bool"
```
where
```
  "inv s ≡ globals_frame_intact s ∧ ipc_buffer_valid s ∧ tls_valid s ∧
           thread_valid s ∧ is_valid_w32 s (Ptr (symbol_table ''swap_a_1'')) ∧
           is_valid_w32 s (Ptr (symbol_table ''swap_a_2'')) ∧
           is_valid_w32 s (Ptr (symbol_table ''swap_b_1'')) ∧
           is_valid_w32 s (Ptr (symbol_table ''swap_b_2'')) ∧
           distinct [symbol_table ''swap_a_1'', symbol_table ''swap_a_2'',
                     symbol_table ''swap_b_1'', symbol_table ''swap_b_2'']"
```

lemma `inv_simps[simp]:`
```
  "inv s ⟹ globals_frame_intact s"
  "inv s ⟹ ipc_buffer_valid s"
  "inv s ⟹ tls_valid s"
  "inv s ⟹ thread_valid s"
  by (simp add:inv_def)+
```

lemma `inv_imp_swap_b_2_valid:`
```
  "inv s ⟹ is_valid_w32 s (Ptr (symbol_table ''swap_b_2''))"
  by (simp only:inv_def)+
```

lemma `inv_imp_swap_a_2_valid:`
```
  "inv s ⟹ is_valid_w32 s (Ptr (symbol_table ''swap_a_2''))"
  by (simp only:inv_def)+
```

lemma `inv_imp_swap_a_1_valid:`
```
  "inv s ⟹ is_valid_w32 s (Ptr (symbol_table ''swap_a_1''))"
  by (simp only:inv_def)+
```

lemma `inv_imp_swap_b_1_valid:`
```
  "inv s ⟹ is_valid_w32 s (Ptr (symbol_table ''swap_b_1''))"
  by (simp only:inv_def)+
```

lemma `inv_imp_swap_a_valid1:`

```
  "⟦p ∈ {Ptr (symbol_table ''swap_a_1''), Ptr (symbol_table ''swap_a_2'')}; inv s⟧
    ⟹ is_valid_w32 s p"
  apply clarsimp
  apply (erule disjE, simp add:inv_imp_swap_a_1_valid)
  apply (simp add:inv_imp_swap_a_2_valid)
  done

lemma inv_imp_swap_a_valid2:
  "⟦p ∈ {Ptr (symbol_table ''swap_a_1''), Ptr (symbol_table ''swap_a_2'')}; inv s⟧
    ⟹ is_valid_w32 s (ptr_coerce p)"
  apply clarsimp
  apply (erule disjE, simp add:inv_imp_swap_a_1_valid)
  apply (simp add:inv_imp_swap_a_2_valid)
  done

lemmas inv_imp_swap_a_valid = inv_imp_swap_a_valid1 inv_imp_swap_a_valid2

lemma inv_imp_swap_b_valid1:
  "⟦p ∈ {Ptr (symbol_table ''swap_b_1''), Ptr (symbol_table ''swap_b_2'')}; inv s⟧
    ⟹ is_valid_w32 s p"
  apply clarsimp
  apply (erule disjE, simp add:inv_imp_swap_b_1_valid)
  apply (simp add:inv_imp_swap_b_2_valid)
  done

lemma inv_imp_swap_b_valid2:
  "⟦p ∈ {Ptr (symbol_table ''swap_b_1''), Ptr (symbol_table ''swap_b_2'')}; inv s⟧
    ⟹ is_valid_w32 s (ptr_coerce p)"
  apply clarsimp
  apply (erule disjE, simp add:inv_imp_swap_b_1_valid)
  apply (simp add:inv_imp_swap_b_2_valid)
  done

lemmas inv_imp_swap_b_valid = inv_imp_swap_b_valid1 inv_imp_swap_b_valid2

lemma inv_imp_distincts:
  "inv s ⟹ distinct [symbol_table ''swap_a_1'', symbol_table ''swap_a_2'',
                     symbol_table ''swap_b_1'', symbol_table ''swap_b_2'']"
  by (simp only:inv_def)
```

The definitions required to fully unfold the invariant.

```
lemmas inv_defs = inv_def globals_frame_intact_def ipc_buffer_valid_def
  ipc_buffer_ptr_def ipc_buffer_def tls_valid_def tls_ptr_def tls_def
  thread_valid_def thread_count_def thread_index_def

lemma seL4_GetIPCBuffer_wp[THEN validNF_make_schematic_post, simplified]:
  "∀s0. ⦃λs. s = s0 ∧
             globals_frame_intact s ∧
             ipc_buffer_valid s⦄
          seL4_GetIPCBuffer'
        ⦃λr s. s = s0 ∧
               r = ipc_buffer_ptr s⦄!"
  apply (rule allI)
```

```
  unfolding seL4_GetIPCBuffer'_def apply wp
  apply (clarsimp simp:inv_defs)
  done

lemma camkes_get_tls_wp[THEN validNF_make_schematic_post, simplified]:
  "∀s0. {|λs. s = s0 ∧ inv s|}
          camkes_get_tls'
        {|λr s. s = s0 ∧ r = tls_ptr s|}!"
  apply (rule allI)
  unfolding camkes_get_tls'_def
  apply (wp seL4_GetIPCBuffer_wp)
  apply (clarsimp simp:tls_ptr_def ipc_buffer_ptr_def)
  done
```

We provide a more abstract monadic definition of `seL4_SetMR` than the AutoCorres-generated version, to ease reasoning.

```
definition
  seL4_SetMR_lifted :: "sword32 ⇒ word32 ⇒ lifted_globals ⇒
                          (unit × lifted_globals) set × bool"
where
  "seL4_SetMR_lifted i val ≡
   do
     ret' ← seL4_GetIPCBuffer';
     guard (λs. i <s seL4_MsgMaxLength);
     guard (λs. 0 <=s i);
     modify (λs. s (|heap_seL4_IPCBuffer__C :=
       (heap_seL4_IPCBuffer__C s)(ret' :=
          msg_C_update (λa. Arrays.update a (unat i) val)
             (heap_seL4_IPCBuffer__C s ret'))
     |))
  od"
```

Show that the abstracted version is actually equivalent to the AutoCorres-generated version.

```
lemma lift_setmr:
  "ipc_buffer_valid s ⟹ seL4_SetMR' i x s = seL4_SetMR_lifted i x s"
  apply (clarsimp simp:seL4_SetMR'_def seL4_SetMR_lifted_def)
  apply monad_eq
  apply (rule conjI)
   apply clarsimp
   apply (rule_tac x=a in exI)
   apply (rule_tac x=b in exI)
   apply (simp cong: lifted_globals.fold_congs)
   apply (simp add:seL4_MsgMaxLength_def)
  apply (rule conjI)
   apply clarsimp
   apply (rule_tac x=a in exI)
   apply (rule_tac x=b in exI)
   apply (clarsimp simp:seL4_MsgMaxLength_def seL4_GetIPCBuffer'_def)
   apply monad_eq
   apply (simp add:inv_defs)
  apply (rule iffI)
   apply clarsimp
   apply (simp add:seL4_GetIPCBuffer'_def)
```

```
   apply monad_eq
   apply (erule disjE)
    apply force
   apply (clarsimp simp:ipc_buffer_valid_def ipc_buffer_ptr_def
                        seL4_MsgMaxLength_def)+
  done
```

A version of the above lemma that is applicable in WP proofs.

```
lemma lift_setmr_hoare:
  "⟦(∀s. P s ⟶ ipc_buffer_valid s); ⦃P⦄ seL4_SetMR_lifted i x ⦃Q⦄!⟧
    ⟹ ⦃P⦄ seL4_SetMR' i x ⦃Q⦄!"
  apply (cut_tac P=P and P'=P and Q=Q and Q'=Q and m="seL4_SetMR' i x" and
                 m'="seL4_SetMR_lifted i x" in validNF_cong)
    apply simp
   apply (subst lift_setmr)
   apply clarsimp
  apply simp+
  done
```

Abstract functional definitions of seL4_SetMR and seL4_GetMR.

```
definition
  setMR :: "lifted_globals ⇒ nat ⇒ word32 ⇒ lifted_globals"
where
  "setMR s i v ≡
      s(|heap_seL4_IPCBuffer__C := (heap_seL4_IPCBuffer__C s)
        (ipc_buffer_ptr s := msg_C_update
          (λa. Arrays.update a i v) (ipc_buffer s))|)"
definition
  getMR :: "lifted_globals ⇒ nat ⇒ word32"
where
  "getMR s i ≡ index (msg_C (ipc_buffer s)) i"
```

Show refinement for seL4_SetMR.

```
lemma seL4_SetMR_wp[wp_unsafe]:
 "⦃λs. globals_frame_intact s ∧
        ipc_buffer_valid s ∧
        0 <=s i ∧
        i <s seL4_MsgMaxLength ∧
        (∀x. P x (setMR s (unat i) v))⦄
    seL4_SetMR' i v
   ⦃P⦄!"
  apply (subst lift_setmr_hoare)
    apply simp_all
  apply (simp add:seL4_SetMR_lifted_def)
  apply (wp seL4_GetIPCBuffer_wp)
  apply (simp add:setMR_def inv_defs del:fun_upd_apply)
  done
```

Show refinement for seL4_GetMR.

```
lemma seL4_GetMR_wp[wp_unsafe]:
  "⦃λs.     0 <=s i ∧
            i <s  seL4_MsgMaxLength ∧
            globals_frame_intact s ∧
```

```
          ipc_buffer_valid s ∧
          P (getMR s (unat i)) s⦄
    seL4_GetMR' i
  ⦃P⦄!"
  apply (simp add:seL4_GetMR'_def)
  apply (wp seL4_GetIPCBuffer_wp)
  apply (simp add:getMR_def inv_defs seL4_MsgMaxLength_def)
  done
```

Some useful properties of the two functional representations.

```
lemma getMR_setMR:
  "⟦i < nat (uint seL4_MsgMaxLength)⟧
    ⟹ getMR (setMR s j v) i = (if i = j then v else getMR s i)"
  apply (simp add:getMR_def setMR_def ipc_buffer_ptr_def ipc_buffer_def
                  fun_upd_def seL4_MsgMaxLength_def)
  done


lemma msg_update_id:"msg_C_update (λr. Arrays.update r i (index (msg_C y) i)) y = y"
  apply (cut_tac f="(λr. Arrays.update r i (index (msg_C y) i))" and
                 f'=id and r=y and r'=y in seL4_IPCBuffer__C_fold_congs(2))
    apply simp+
  apply (simp add:id_def)
  apply (metis seL4_IPCBuffer__C_accupd_diff(22) seL4_IPCBuffer__C_accupd_diff(24)
               seL4_IPCBuffer__C_accupd_diff(26) seL4_IPCBuffer__C_accupd_diff(28)
               seL4_IPCBuffer__C_accupd_diff(30) seL4_IPCBuffer__C_accupd_diff(41)
               seL4_IPCBuffer__C_accupd_same(2) seL4_IPCBuffer__C_idupdates(1))
  done

lemma setMR_getMR:
  "setMR s i (getMR s i) = s"
  apply (simp add:setMR_def getMR_def ipc_buffer_ptr_def ipc_buffer_def)
  apply (subst msg_update_id)
  apply simp
  done

lemma setMR_setMR[simp]:"setMR (setMR s i x) i y = setMR s i y"
  apply (simp add:setMR_def ipc_buffer_ptr_def ipc_buffer_def fun_upd_def)
  apply (subst comp_def, subst update_update)
  apply simp
  done

lemma setMR_reorder:
  "setMR (setMR s i x) j y = (if i = j then setMR s j y else setMR (setMR s j y) i x)"
  apply simp
  apply (simp add:setMR_def ipc_buffer_ptr_def ipc_buffer_def fun_upd_def comp_def)
  done

lemma setMR_preserves_inv[simp]:"inv s ⟹ inv (setMR s i x)"
  apply (clarsimp simp:inv_defs setMR_def)
  done

lemma tls_mr_distinct[simp]:"tls (setMR s x y) = tls s"
```

```
  apply (clarsimp simp:tls_def tls_ptr_def setMR_def ipc_buffer_ptr_def
                       ipc_buffer_def)
  done

definition
  packed' :: "lifted_globals ⇒ word32 list ⇒ bool"
where
  "packed' s xs =
    (length xs ≤ unat seL4_MsgMaxLength ∧
     list_all (λ(i, v). getMR s i = v) (enumerate 0 xs))"

lemma packed_empty[simp]: "packed' s []"
  by (simp add:packed'_def)

lemma packed_eq:"⟦packed' s xs; packed' s ys; length xs = length ys⟧ ⟹ xs = ys"
  apply (rule nth_equalityI)
   apply assumption
  unfolding packed'_def apply clarsimp
  apply (drule_tac i=i in list_all_nth, simp)+
  apply (simp add:split_beta fst_enumerate snd_enumerate)
  done

lemma packed_length:"packed' s xs ⟹ length xs ≤ unat seL4_MsgMaxLength"
  by (simp add:packed'_def)

lemma packed_setmr[simp]:"packed' (setMR s 0 x) [x]"
  apply (clarsimp simp:packed'_def getMR_setMR seL4_MsgMaxLength_def)
  done

lemma getmr_packed:"⟦packed' s xs; i < length xs⟧ ⟹ getMR s i = xs ! i"
  apply (clarsimp simp:packed'_def)
  apply (subst (asm) list_all_iff)
  apply (erule_tac x="(i, xs ! i)" in ballE)
   apply clarsimp
  apply (subgoal_tac "(i, xs ! i) ∈ set (enumerate 0 xs)")
   apply simp
  apply (rule enumerate_member[where n=0, simplified])
  apply simp
  done

lemma packed_extend:
  "⋀x. ⟦i < unat seL4_MsgMaxLength; i = length xs; packed' s xs⟧
    ⟹ packed' (setMR s i x) (xs @ [x])"
  apply (clarsimp simp:packed'_def)
  apply (subst enumerate_append)
  apply (subst list_all_append)
  apply clarsimp
  apply (cut_tac s=s and i="length xs" and j="length xs" and v=x in getMR_setMR)
   apply (clarsimp simp:seL4_MsgMaxLength_def)+
  apply (cut_tac xs="enumerate 0 xs" and ys = "enumerate 0 xs" and
                 f="λ(i, y). getMR (setMR s (length xs) x) i = y" and
                 g="λ(i, y). getMR s i = y" in list_all_cong)
    apply (clarsimp simp:seL4_MsgMaxLength_def)+
```

```
   apply (subst getMR_setMR)
    apply (subgoal_tac "a < length xs")
     apply (clarsimp simp:seL4_MsgMaxLength_def)
    apply (rule_tac b=b in enumerate_bound[where n=0, simplified])
    apply assumption
   apply (clarsimp simp:enumerate_exceed[where n=0, simplified])
  apply clarsimp
  done

lemma packed_equiv:
  "⋀x. ⟦i < unat seL4_MsgMaxLength; i = length xs⟧
    ⟹ packed' s xs = packed' (setMR s i x) (xs @ [x])"
  apply (rule iffI)
   apply (rule packed_extend, clarsimp+)
  apply (clarsimp simp:packed'_def)
  apply (subst (asm) enumerate_append)
  apply (subst (asm) list_all_append)
  apply clarsimp
  apply (cut_tac xs="enumerate 0 xs" and ys="enumerate 0 xs" and
                 f="λ(i, v). getMR s i = v" and
                 g="λ(i, y). getMR (setMR s (length xs) x) i = y" in list_all_cong)
    apply clarsimp+
   apply (drule_tac b=b in enumerate_bound[where n=0, simplified])
   apply (subst getMR_setMR)
    apply (clarsimp simp:seL4_MsgMaxLength_def)+
  done

definition
  ptr_valid_s32 :: "lifted_globals ⇒ 32 sword ptr ⇒ bool"
where
  "ptr_valid_s32 s p ≡ is_valid_w32 s (ptr_coerce p)"

definition
  ptr_contains_s32 :: "lifted_globals ⇒ 32 sword ptr ⇒ 32 sword ⇒ bool"
where
  "ptr_contains_s32 s p v ≡ ptr_valid_s32 s p ∧ heap_w32 s (ptr_coerce p) = scast v"

lemma ptr_contains_s32_valid[simp]:
  "ptr_contains_s32 s p v ⟹ ptr_valid_s32 s p"
  by (clarsimp simp:ptr_contains_s32_def)

lemma ptr_contains_s32_setmr[simp]:
  "ptr_contains_s32 (setMR s i x) p v = ptr_contains_s32 s p v"
  by (clarsimp simp:ptr_contains_s32_def ptr_valid_s32_def setMR_def)

lemma ptr_valid_s32_setmr[simp]:
  "ptr_valid_s32 (setMR s i x) p = ptr_valid_s32 s p"
  by (clarsimp simp:ptr_valid_s32_def setMR_def)

definition
  update_s32 :: "lifted_globals ⇒ 32 sword ptr ⇒ 32 sword ⇒ lifted_globals"
where
  "update_s32 s p v ≡
```

```
    heap_w32_update (λf q. if q = ptr_coerce p then scast v else f q) s"

lemma setmr_update_s32_reorder[simp]:
  "setMR (update_s32 s p v) i x = update_s32 (setMR s i x) p v"
  by (simp add:setMR_def update_s32_def ipc_buffer_def ipc_buffer_ptr_def)

lemma heap_w32_update_preserves_inv[simp]:
  "inv s ⟹ inv (heap_w32_update f s)"
  by (simp add:inv_defs)

lemma heap_w32_update_equiv:
  "⟦f = g; s = s'⟧ ⟹ heap_w32_update f s = heap_w32_update g s'"
  by clarsimp

lemma heap_w32_update_compose:
  "heap_w32_update (f ∘ g) s = heap_w32_update f (heap_w32_update g s)"
  by clarsimp

lemma getmr_heap_w32_update[simp]:
  "getMR (heap_w32_update f s) i = getMR s i"
  by (simp add:getMR_def ipc_buffer_def ipc_buffer_ptr_def)

lemma setmr_heap_w32_update_reorder:
  "setMR (heap_w32_update f s) i x = heap_w32_update f (setMR s i x)"
  by (simp add:setMR_def ipc_buffer_def ipc_buffer_ptr_def)

lemma heap_w32_update_preserves_ptr_valid_s32[simp]:
  "ptr_valid_s32 s p ⟹ ptr_valid_s32 (heap_w32_update f s) p"
  by (simp add:ptr_valid_s32_def)

lemma of_int_via_ucast_32[simp]:
  "((ucast ((of_int x)::32 word))::32 sword) = of_int x"
  by simp

lemma update_s32_preserves_inv[simp]:
  "inv s ⟹ inv (update_s32 s p v)"
  by (simp add:update_s32_def)

lemma is_valid_w32_update_s32[simp]:
  "is_valid_w32 (update_s32 s p v) = is_valid_w32 s"
  by (simp add:update_s32_def)

lemma heap_w32_setmr[simp]: "heap_w32 (setMR s i x) = heap_w32 s"
  by (simp add:setMR_def)

lemma heap_w32_update_s32:
  "heap_w32 (update_s32 s p v) q = (if (ptr_coerce p) = q then scast v else heap_w32 s q)"
  by (simp add:update_s32_def)

lemma get_swap_a_wp[THEN validNF_make_schematic_post, simplified]:
  "∀s0. {|λs. s = s0 ∧ inv s|}
          get_swap_a'
        {|λr s. s = s0 ∧ r ∈ {Ptr (symbol_table ''swap_a_1''),
```

```
                          Ptr (symbol_table ''swap_a_2'')} ∧
              inv s⦈!"
  apply (rule allI)
  unfolding get_swap_a'_def apply (wp camkes_get_tls_wp)
  apply clarsimp
  apply (frule inv_simps(3), unfold tls_valid_def, simp)
  apply (frule inv_simps(4),
         unfold thread_valid_def thread_index_def thread_count_def tls_def)
  apply (subgoal_tac "is_valid_w32 s0 (Ptr (symbol_table ''swap_a_1''))",
         subgoal_tac "is_valid_w32 s0 (Ptr (symbol_table ''swap_a_2''))")
    apply unat_arith
   apply (simp only:inv_def)+
  done

lemma get_swap_b_wp[THEN validNF_make_schematic_post, simplified]:
  "∀s0. ⦇λs. s = s0 ∧ inv s⦈
          get_swap_b'
        ⦇λr s. s = s0 ∧ r ∈ {Ptr (symbol_table ''swap_b_1''),
                          Ptr (symbol_table ''swap_b_2'')} ∧
              inv s⦈!"
  apply (rule allI)
  unfolding get_swap_b'_def apply (wp camkes_get_tls_wp)
  apply clarsimp
  apply (frule inv_simps(3), unfold tls_valid_def, simp)
  apply (frule inv_simps(4),
         unfold thread_valid_def thread_index_def thread_count_def tls_def)
  apply (subgoal_tac "is_valid_w32 s0 (Ptr (symbol_table ''swap_b_1''))",
         subgoal_tac "is_valid_w32 s0 (Ptr (symbol_table ''swap_b_2''))")
    apply unat_arith
   apply (simp only:inv_def)+
  done

end
```

Extend the C locale with assumptions on the user's code.

```
locale conn_glue = conn +
  fixes swap_P :: "lifted_globals ⇒ 32 sword ⇒ 32 sword ⇒ bool"
  fixes swap_Q :: "lifted_globals ⇒ lifted_globals ⇒ 32 word ⇒ 32 sword ⇒
                   32 sword ⇒ 32 sword ⇒ 32 sword ⇒ bool"
  assumes ss_swap_wp':
    "∀s0 p0_in p1_in. ⦇λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0 ∧
                           p0_in = ucast (heap_w32 s (ptr_coerce p0)) ∧
                           (p0 = Ptr (symbol_table ''swap_a_1'') ∨
                            p0 = Ptr (symbol_table ''swap_a_2'')) ∧
                           ptr_valid_s32 s p1 ∧
                           p1_in = ucast (heap_w32 s (ptr_coerce p1)) ∧
                           (p1 = Ptr (symbol_table ''swap_b_1'') ∨
                            p1 = Ptr (symbol_table ''swap_b_2'')) ∧
                           swap_P s p0_in p1_in⦈
                      ss_swap' p0 p1
                     ⦇λr s. inv s ∧ (∃p0_out p1_out.
                               p0_out = ucast (heap_w32 s (ptr_coerce p0)) ∧
                               p1_out = ucast (heap_w32 s (ptr_coerce p1)) ∧
```

```
                                swap_Q s0 s r p0_in p0_out p1_in p1_out)⦘!"
  assumes swap_pre_stable_setmr[simp]:
    "⋀s i x. swap_P (setMR s i x) = swap_P s"
  assumes swap_post_stable_setmr1[simp]:
    "⋀ s i x. swap_Q (setMR s i x) = swap_Q s"
  assumes swap_post_stable_setmr2[simp]:
    "⋀ s s' i x. swap_Q s (setMR s' i x) = swap_Q s s'"
  assumes swap_pre_stable_update_a[simp]:
    "⋀v s p. ⟦p = Ptr (symbol_table ''swap_a_1'') ∨
              p = Ptr (symbol_table ''swap_a_2'')⟧
      ⟹ swap_P (update_s32 s p v) = swap_P s"
  assumes swap_post_stable_update_a[simp]:
    "⋀v s p. ⟦p = Ptr (symbol_table ''swap_a_1'') ∨
              p = Ptr (symbol_table ''swap_a_2'')⟧
      ⟹ swap_Q (update_s32 s p v) = swap_Q s"
  assumes swap_pre_stable_update_b[simp]:
    "⋀v s p. ⟦p = Ptr (symbol_table ''swap_b_1'') ∨
              p = Ptr (symbol_table ''swap_b_2'')⟧
      ⟹ swap_P (update_s32 s p v) = swap_P s"
  assumes swap_post_stable_update_b[simp]:
    "⋀v s p. ⟦p = Ptr (symbol_table ''swap_b_1'') ∨
              p = Ptr (symbol_table ''swap_b_2'')⟧
      ⟹ swap_Q (update_s32 s p v) = swap_Q s"
  assumes ptr_valid_s32_stable_swap:
    "⋀s s' r p0_in p0_out p1_in p1_out.
      swap_Q s s' r p0_in p0_out p1_in p1_out
        ⟹ ptr_valid_s32 s' = ptr_valid_s32 s"
  assumes swap_a_1_swap_a_2_distinct[simplified eq_commute, simp]:
    "symbol_table ''swap_a_1'' ≠ symbol_table ''swap_a_2''"
  assumes swap_a_1_swap_b_1_distinct[simplified eq_commute, simp]:
    "symbol_table ''swap_a_1'' ≠ symbol_table ''swap_b_1''"
  assumes swap_a_1_swap_b_2_distinct[simplified eq_commute, simp]:
    "symbol_table ''swap_a_1'' ≠ symbol_table ''swap_b_2''"
  assumes swap_a_2_swap_b_1_distinct[simplified eq_commute, simp]:
    "symbol_table ''swap_a_2'' ≠ symbol_table ''swap_b_1''"
  assumes swap_a_2_swap_b_2_distinct[simplified eq_commute, simp]:
    "symbol_table ''swap_a_2'' ≠ symbol_table ''swap_b_2''"
  assumes swap_b_1_swap_b_2_distinct[simplified eq_commute, simp]:
    "symbol_table ''swap_b_1'' ≠ symbol_table ''swap_b_2''"
begin

lemma cs_swap_marshal_wp[THEN validNF_make_schematic_post, simplified]:
  "∀s0. ⦃λs. inv s ∧ s0 = setMR (setMR (setMR s 0 0) 1 (scast p0)) 2 (scast p1)⦄
          cs_swap_marshal' p0 p1
        ⦃λr s. s = s0 ∧ inv s ∧ r = 3 ∧ packed' s [0, scast p0, scast p1]⦄!"
  apply (rule allI)
  unfolding cs_swap_marshal'_def
  apply (wp seL4_SetMR_wp)
  apply (clarsimp simp:packed'_def seL4_MsgMaxLength_def)
  apply (subst getMR_setMR, simp add:seL4_MsgMaxLength_def)+
  apply (simp add:cast_down_s64 cast_down_u64)
  done
```

```
lemma ss_swap_unmarshal_wp[THEN all_pair_unwrap[THEN iffD2],
                           THEN all_pair_unwrap[THEN iffD2],
                           THEN validNF_make_schematic_post, simplified]:
  "∀s0 p0' p1'. {|λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0 ∧ ptr_valid_s32 s p1 ∧
                      distinct [((ptr_coerce p0)::32 word ptr),
                                ((ptr_coerce p1)::32 word ptr)] ∧
                      packed' s [0, p0', p1']|}
          ss_swap_unmarshal' p0 p1
        {|λ_ s. inv s ∧ ptr_contains_s32 s p0 (ucast p0') ∧
                ptr_contains_s32 s p1 (ucast p1') ∧
                s = update_s32 (update_s32 (s0) p0 (ucast p0')) p1 (ucast p1')|}!"
  apply (rule allI)+
  unfolding ss_swap_unmarshal'_def
  apply (wp seL4_GetMR_wp)
  apply (clarsimp simp:seL4_MsgMaxLength_def ptr_valid_s32_def
                       ptr_contains_s32_def update_s32_def)
  apply (clarsimp simp:getmr_packed ucast_id)
  apply (((rule heap_w32_update_equiv), rule ext, rule ext,
             simp add:getmr_packed ucast_id)+)?
  apply simp
  done

lemma ss_swap_invoke_subst:
  "ss_swap_invoke' = ss_swap'"
  by (clarsimp simp:ss_swap_invoke'_def ss_swap'_def intro!:ext)

lemmas ss_swap_wp = ss_swap_wp'[THEN all_pair_unwrap[THEN iffD2],
                                THEN all_pair_unwrap[THEN iffD2],
                                THEN validNF_make_schematic_post, simplified]

lemma ss_swap_marshal_wp[THEN validNF_make_schematic_post, simplified]:
  "∀s0. {|λs. s = s0 ∧ inv s|}
          ss_swap_marshal' ret p0 p1
        {|λr s. r = 3 ∧ s = setMR (setMR (setMR (s0) 0 ret) 1 (scast p0)) 2 (scast p1) ∧
                packed' s [ret, scast p0, scast p1]|}!"
  apply (rule allI)
  unfolding ss_swap_marshal'_def
  apply (wp seL4_SetMR_wp)
  apply (clarsimp simp:packed'_def seL4_MsgMaxLength_def getMR_setMR)
  done

lemma ss_swap_internal_wp[THEN all_pair_unwrap[THEN iffD2],
                          THEN all_pair_unwrap[THEN iffD2],
                          THEN validNF_make_schematic_post, simplified]:
  "∀s0 p0 p1. {|λs. s = s0 ∧ inv s ∧ packed' s [0, scast p0, scast p1] ∧ swap_P s p0 p1|}
          ss_swap_internal'
        {|λr s. inv s ∧ (∃ru p0_out p1_out.
                  packed' s [ru, scast p0_out, scast p1_out] ∧
                  swap_Q s0 s ru p0 p0_out p1 p1_out)|}!"
  apply (rule allI)+
  unfolding ss_swap_internal'_def
  apply (subst ss_swap_invoke_subst)
  apply (wp ss_swap_unmarshal_wp ss_swap_wp ss_swap_marshal_wp get_swap_a_wp
```

```
          get_swap_b_wp)
apply clarsimp

apply (subgoal_tac "rv = Ptr (symbol_table ''swap_a_1'') ∨
                    rv = Ptr (symbol_table ''swap_a_2'')
                      ⟶ ((ptr_coerce rv)::32 word ptr) ≠ Ptr (symbol_table ''swap_b_1'')")
 prefer 2
 apply fastforce
apply (subgoal_tac "rv = Ptr (symbol_table ''swap_a_1'') ∨
                    rv = Ptr (symbol_table ''swap_a_2'')
                      ⟶ ((ptr_coerce rv)::32 word ptr) ≠ Ptr (symbol_table ''swap_b_2'')")
 prefer 2
 apply fastforce
apply clarsimp
apply (clarsimp simp:inv_imp_swap_a_valid inv_imp_swap_a_1_valid inv_imp_swap_a_2_valid
                     ptr_valid_s32_def inv_imp_swap_b_valid inv_imp_swap_b_1_valid
                     inv_imp_swap_b_2_valid)
apply (rule conjI)
 apply clarsimp
 apply ((rule conjI, clarsimp)+)?

 apply (rule_tac x="scast p0" in exI)
 apply (rule_tac x="scast p1" in exI)
 apply clarsimp

 apply (rule_tac x=p0 in exI)
 apply (rule conjI)
  apply (clarsimp simp:heap_w32_update_s32)
 apply (rule_tac x=p1 in exI)
 apply (rule conjI)
  apply (clarsimp simp:heap_w32_update_s32)

 apply (rule conjI)
  apply (clarsimp simp:ptr_contains_s32_def, fastforce?)[1]
 apply clarsimp

 apply (rule_tac x=rva in exI)
 apply (rule_tac x="ucast (heap_w32 s' (ptr_coerce rv))" in exI)
 apply (rule_tac x="ucast (heap_w32 s' (ptr_coerce
                      (Ptr (symbol_table ''swap_b_1'')))))" in exI)

 apply (clarsimp simp:ptr_contains_s32_def)
apply clarsimp
apply ((rule conjI, clarsimp)+)?

apply (rule_tac x="scast p0" in exI)
apply (rule_tac x="scast p1" in exI)
apply clarsimp

apply (rule_tac x=p0 in exI)
apply (rule conjI)
 apply (clarsimp simp:heap_w32_update_s32)
apply (rule_tac x=p1 in exI)
```

```
  apply (rule conjI)
   apply (clarsimp simp:heap_w32_update_s32)

  apply (rule conjI)
   apply (clarsimp simp:ptr_contains_s32_def, fastforce?)[1]
  apply clarsimp

  apply (rule_tac x=rva in exI)
  apply (rule_tac x="ucast (heap_w32 s' (ptr_coerce rv))" in exI)
  apply (rule_tac x="ucast (heap_w32 s' (ptr_coerce
                      (Ptr (symbol_table ''swap_b_2'')))))" in exI)

  apply (clarsimp simp:ptr_contains_s32_def)
  done

lemma cs_swap_unmarshal_wp[THEN all_pair_unwrap[THEN iffD2],
                           THEN all_pair_unwrap[THEN iffD2],
                           THEN all_pair_unwrap[THEN iffD2],
                           THEN validNF_make_schematic_post, simplified]:
  "∀s0 ret p0' p1'. {|λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0 ∧ ptr_valid_s32 s p1 ∧
                        distinct [((ptr_coerce p0)::32 word ptr),
                                  ((ptr_coerce p1)::32 word ptr)] ∧
                        packed' s [ret, p0', p1']|}
         cs_swap_unmarshal' p0 p1
       {|λr s. inv s ∧ r = ret ∧ ptr_contains_s32 s p0 (ucast p0') ∧
               ptr_contains_s32 s p1 (ucast p1') ∧
               s = update_s32 (update_s32 (s0) p0 (ucast p0')) p1 (ucast p1')|}!"
  apply (rule allI)+
  unfolding cs_swap_unmarshal'_def
  apply (wp seL4_GetMR_wp)
  apply (clarsimp simp:packed'_def seL4_MsgMaxLength_def ucast_id ptr_contains_s32_def
                      update_s32_def ptr_valid_s32_def)
  apply (simp add:numeral_2_eq_2 cong del:if_weak_cong)?
  done

lemma cs_ss_swap_equiv_wp:
  "∀s0. {|λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0_out ∧ ptr_valid_s32 s p1_out ∧
              distinct [((ptr_coerce p0_out)::32 word ptr),
                        ((ptr_coerce p1_out)::32 word ptr)] ∧
              (∀s1 s2 v. swap_Q s1 (update_s32 s2 p0_out v) = swap_Q s1 s2) ∧
              (∀s1 s2 v. swap_Q s1 (update_s32 s2 p1_out v) = swap_Q s1 s2) ∧
              swap_P s p0 p1|}
         do cs_swap_marshal' p0 p1;
            ss_swap_internal';
            cs_swap_unmarshal' p0_out p1_out
         od
       {|λr s. inv s ∧ swap_Q s0 s r p0 (ucast (heap_w32 s (ptr_coerce p0_out)))
                         p1 (ucast (heap_w32 s (ptr_coerce p1_out)))|}!"
  apply (rule allI)+
  apply (wp cs_swap_unmarshal_wp
            ss_swap_internal_wp
            cs_swap_marshal_wp)
  apply clarsimp
```

```
  apply (rule_tac x=p0 in exI)
  apply (rule_tac x=p1 in exI)
  apply (clarsimp simp:ptr_valid_s32_stable_swap)
  apply (rule_tac x=ru in exI)
  apply (rule_tac x="scast p0_outa" in exI)
  apply (rule_tac x="scast p1_outa" in exI)
  apply (clarsimp simp:heap_w32_update_s32)
  done

end

end
```