Formal specification of the Cap9 kernel

Mikhail Mandrykin Ilya Shchepetkov

May 17, 2019

Contents

1 Introduction

This is an Isabelle/HOL theory that describes and proves the correctness of the Cap9 kernel specification.

2 Preliminaries

```
theory Cap9
imports
  "HOL−Word.Word"
begin
```

We start with some types and definitions that will be used later.

2.1 Procedure keys

Procedure keys are represented as 24-byte (192 bits) machine words. Keys will be used both in the abstract and concrete states.

```
type_synonym word24 = "192 word"
type_synonym key = word24
```

Instantiate *len0* type class to extract lengths from the *key* and other word types avoiding repeated explicit numeric specification of the length.

```
instantiation word :: (len0) len0 begin
definition len_word[simp]: "len_of (_ :: 'a::len0 word itself) = LENGTH('a)"
instance ..
end
```

We make some assumptions about the set of all procedures that can be registered in the system:

1. there is a hash function that maps the set of all procedures to the set of all keys;
2. this function is injective on the set;
3. number of all procedures is smaller or equal to the number of all keys.

The assumptions concretize our hypothesis about the absence of procedure key collisions. We formalize these assumptions by defining a corresponding Isar type class of allowed procedures:

```
class proc_class =
  fixes key :: "'a ⇒ key"
  assumes "CARD ('a) ≤ CARD (key)"
  assumes key_inj:"inj key"
```

To insure we don't introduce contradictions with these assumptions we build a sample model of the set of all procedures. Although here use a relatively simple hash function, we don't impose any additional requirements on the function, so it can be replaced with any real cryptographic hash. We only need a single procedure-key pair to be calculated in advance, which is easily achievable by computing the hash of some trivial allowed procedure.

We proceed with the corresponding definitions.

2.2 Dummy hash function

Byte is 8-bit machine word:

```
type_synonym byte = "8 word"
```

As an example we use a simple *djb2* hash function to compute 24-byte hash of a list of bytes.

```
abbreviation "seed :: key ≡ 5381"
fun djb2 :: "byte list ⇒ key" where
  "djb2 []    =    seed" |
  "djb2 (e # es)   =   (let h = djb2 es in (h << 5) + h + ucast e)"
```

To formalize a notion of a set of procedures without hash collisions we nonconstructively define a choice function to select exactly one arbitrary procedure for each possible key. In reality this corresponds to the assumption that we never encounter hash collisions, so the choice function can be assumed to be always well-defined on the current set of procedure keys since the Hilbert epsilon operator's choice is arbitrary.

```
definition "choose_proc k ≡
  if k = seed then
    {[]}
  else if ∃ p. djb2 p = k then
    {SOME p. djb2 p = k}
  else
    {}"
```

```
lemma choose_proc[simp]: "x ∈ choose_proc k ⟹ djb2 x = k"
  unfolding choose_proc_def
  by (auto split: if_splits intro: someI)
```

The procedure corresponding to each key is unique.

```
lemma[simp]: "⟦x ∈ choose_proc k; y ∈ choose_proc k⟧ ⟹ x = y"
  unfolding choose_proc_def
  by (simp split: if_splits)
```

2.3 Dummy set of procedures

Procs is a set of all allowed procedures, without procedure key collisions:

```
definition "Procs ≡ ⋃k. choose_proc k"
```

A new type *proc* is the sought instantiation of the *proc_class* type class.

```
typedef proc = Procs
  unfolding Procs_def choose_proc_def
  by (rule exI[of _ "[]"], auto)
```

2.3.1 Injectivity of the hash function

Hash function is injective on the domain of all procedures.

```
lemma djb2_inj: "inj_on djb2 Procs"
  unfolding inj_on_def Procs_def
  by auto
```

2.3.2 Number of all procedures

Here we introduce maximum number of registered procedure keys:

```
abbreviation "max_nkeys ≡ 2 ^ LENGTH(key) :: nat"
```

Number of all procedures must be equal or smaller then the maximum number of procedure keys.

```
lemma card_procs: "card Procs ≤ max_nkeys"
  unfolding Procs_def
proof (subst card_UN_disjoint)
  show "finite (UNIV :: key set)"
    and "∀i∈UNIV. finite (choose_proc i)"
    unfolding choose_proc_def
    by (simp_all split: if_splits)
  show "∀i∈UNIV. ∀j∈UNIV. i ≠ j ⟶ choose_proc i ∩ choose_proc j = {}"
    by (auto, drule choose_proc, simp)
  show "(∑i∈UNIV. card (choose_proc i)) ≤ max_nkeys"
    using sum_bounded_above[of "UNIV :: key set" "λ i. card (choose_proc i)", where K = 1]
    unfolding choose_proc_def card_word
    by auto
qed
```

2.3.3 Instantiation

Here we show that there the dummy type *proc* satisfies our assumptions.

```
instantiation proc :: proc_class
begin
definition "key ≡ djb2 ∘ Rep_proc"

instance proof
  have "CARD(key) = 2 ^ LENGTH(key)" by (simp add:card_word)
  thus "CARD(proc) ≤ CARD(key)"
    using card_procs
          type_definition.card[OF proc.type_definition_proc]
    by auto
  show "inj (key :: proc ⇒ _)"
    using djb2_inj proc.Rep_proc proc.Rep_proc_inject
    unfolding inj_def key_proc_def inj_on_def
    by force
qed
end
```

3 Abstract state

Abstract state is implemented as a record with a single component labeled "procs". This component is a mapping from the set of procedure keys to the direct product of procedure indexes and procedure data.

```
record ('p :: proc_class) abs =
  procs    :: "key ⇀ nat × 'p"
```

3.1 Abbreviations

Here we introduce some useful abbreviations that will simplify the expression of the abstract state properties.

Number of the procedures in the abstract state:

```
abbreviation "nprocs σ ≡ card (dom (procs σ))"
```

List of procedure keys:

```
abbreviation "proc_keys σ ≡ dom (procs σ)"
```

List of procedure indexes:

```
abbreviation "proc_ids σ ≡ {1..nprocs σ}"
```

Pair with the procedure index and procedure itself for a given key:

```
abbreviation "proc σ k ≡ the (procs σ k)"
```

Procedure index for a given key:

```
abbreviation "proc_id σ k ≡ fst (proc σ k)"
```

Procedure itself for a given key:

```
abbreviation "proc_bdy σ k ≡ snd (proc σ k)"
```

Maximum number of procedures in the abstract state:

```
abbreviation "proc_id_len ≡ 24"
abbreviation "max_nprocs ≡ 2 ^ proc_id_len - 1 :: nat"
```

3.1.1 Well-formedness

For each procedure key the following must be true:

1. corresponding procedure index on the interval from 1 to the number of procedures in the state;
2. key is a valid hash of the procedure data;
3. number of procedures in the state is smaller or equal to the maximum number.

definition *"procs_rng_wf σ ≡*
(∀ k ∈ proc_keys σ. proc_id σ k ∈ proc_ids σ ∧ key (proc_bdy σ k) = k) ∧
nprocs σ ≤ max_nprocs"

Procedure indexes must be injective.

definition *"procs_map_wf σ ≡ inj_on (proc_id σ) (proc_keys σ)"*

Abstract state is well-formed if the previous two properties are satisfied.

definition *abs_wf :: "'p :: proc_class abs ⇒ bool" ("⊢ _" [60] 60)* **where**
"⊢σ ≡
procs_rng_wf σ
∧ procs_map_wf σ"

lemmas *procs_rng_wf = abs_wf_def procs_rng_wf_def*

lemmas *procs_map_wf = abs_wf_def procs_map_wf_def*

4 Storage state

32-byte machine words that are used to model keys and values of the storage.

type_synonym *word32 = "256 word"*

Storage is a function that takes a 32-byte word (key) and returns another 32-byte word (value).

type_synonym *storage = "word32 ⇒ word32"*

Storage key that corresponds to the number of procedures in the list:

abbreviation *nprocs_addr ("@nprocs")* **where** *"nprocs_addr ≡ 0xffffffff01 << (27 ∗ 8) :: word32"*

Storage key that corresponds to the procedure key with index i:

definition *proc_key_addr ("@proc'_key")* **where** *"@proc_key i ≡ @nprocs OR of_nat i"*

Procedure index that corresponds to some procedure key address:

definition *id_of_proc_key_addr* **where** *"id_of_proc_key_addr a ≡ unat (@nprocs XOR a)"*

Maximum number of procedures in the kernel, but in the form of a 32-byte machine word:

abbreviation *"max_nprocs_word ≡ 2 ^ proc_id_len − 1 :: word32"*

Declare some lemmas as simplification rules:

declare *unat_word_ariths[simp] word_size[simp]*

Storage address that corresponds to the procedure heap for a given procedure key:

abbreviation *"proc_heap_mask ≡ 0xffffffff00 << (27 ∗ 8) :: word32"*
abbreviation *proc_heap_addr :: "key ⇒ word32" ("@proc'_heap")* **where**
"@proc_heap k ≡ proc_heap_mask OR ((ucast k) << (3 ∗ 8))"

Storage address that corresponds to the procedure address:

abbreviation *proc_addr_addr ("@proc'_addr")* **where** *"@proc_addr k ≡ @proc_heap k"*

Storage address that corresponds to the procedure index:

```
abbreviation proc_id_addr ("@proc'_id") where "@proc_id k ≡ @proc_heap k OR 0x01"
```

Procedure key that corresponds to some procedure index address:

```
abbreviation proc_key_of_id_addr :: "word32 ⇒ key" where
  "proc_key_of_id_addr a ≡ ucast (proc_heap_mask XOR a)"
```

Storage address that corresponds to the number of capabilities of type t:

```
abbreviation ncaps_addr :: "key ⇒ byte ⇒ word32" ("@ncaps") where
  "@ncaps k t ≡ @proc_heap k OR (ucast t << 2 ∗ 8)"
```

Storage address that corresponds to the capability of type t, with index $i - 1$, and offset *off* into that capability:

```
abbreviation proc_cap_addr :: "key ⇒ byte ⇒ byte ⇒ byte ⇒ word32" ("@proc'_cap") where
  "@proc_cap k t i off ≡ @proc_heap k OR (ucast t << 2 ∗ 8) OR (ucast i << 8) OR ucast off"
```

4.1 Lemmas

4.1.1 Auxiliary lemmas about procedure key addresses

Valid procedure id has all zeros in its higher bits.

```
lemma proc_id_high_zeros[simp]:
  "n ≤ max_nprocs_word =⇒ ∀ i∈{proc_id_len..<LENGTH(word32)}. ¬ n !! i"
  (is "?nbound =⇒ ∀ _ ∈ ?high. _")
proof
  fix i
  assume 0:"i ∈ ?high"
  from 0 have "2 ^ proc_id_len ≤ (2 :: nat) ^ i" by (simp add: numerals(2))
  moreover from 0 have "0 < (2 :: word32) ^ i" by (subst word_2p_lem; simp)
  ultimately have "2 ^ proc_id_len ≤ (2 :: word32) ^ i"
    unfolding word_le_def
    by (subst (asm) of_nat_le_iff[symmetric], simp add:uint_2p)
  thus "?nbound =⇒ ¬ n !! i"
    unfolding not_def
    by (intro impI) (drule bang_is_le, unat_arith)
qed
```

Address of the # of procedures has all zeros in its lower bits.

```
lemma nprocs_key_low_zeros[simp]: "∀ i∈{0..<proc_id_len}. ¬ @nprocs !! i"
  by (subst nth_shiftl, auto)
```

Elimination (generalized split) rule for 32-byte words: a property holds on all bits if and only if it holds on the higher and lower bits.

```
lemma low_high_split:
   "(∀ n. P ((x :: word32) !! n)) =
    ((∀ n∈{0..<proc_id_len}. P (x !! n)) ∧
     (∀ n∈{proc_id_len..<LENGTH(word32)}. P (x !! n)) ∧
     P False)"
  (is "?left = ?right")
proof (intro iffI)
  have "¬ x !! size x" using test_bit_size[of x "size x"] by blast
  hence "?left =⇒ P False" by (metis (full_types))
  thus "?left =⇒ ?right" by auto

  show "?right =⇒ ?left" using test_bit_size[of x] by force
qed
```

Computing procedure key address by its id is an invertible operation.

```
lemma id_of_key_addr_inv[simp]:
  "i ≤ max_nprocs ⟹ id_of_proc_key_addr (@proc_key i) = i"
  (is "?ibound ⟹ ?rev")
proof-
  assume 0:"?ibound"
  hence 1:"unat (of_nat i :: word32) = i"
    by (simp add: le_unat_uoi[where z=max_nprocs_word])
  hence "of_nat i ≤ max_nprocs_word"
    using 0
    by (simp add: word_le_nat_alt)
  hence "@nprocs XOR @nprocs OR (of_nat i) = of_nat i"
    using nprocs_key_low_zeros proc_id_high_zeros
    by (auto simp add: word_eq_iff word_ops_nth_size)
  thus "?rev"
    using 1
    unfolding proc_key_addr_def id_of_proc_key_addr_def
    by simp
qed
```

5 Correspondence between abstract and storage states

Number of procedures is stored by the corresponding address (*@nprocs*).

```
definition "models_nprocs s σ ≡ unat (s @nprocs) = nprocs σ"
```

Each procedure key k is stored by the corresponding address (*@proc_key* k).

```
definition "models_proc_keys s σ ≡
  ∀ k ∈ proc_keys σ. s (@proc_key (proc_id σ k)) = ucast k"
```

For each procedure key k its index is stored by the corresponding address (*@proc_id* k).

```
definition "models_proc_ids s σ ≡
  ∀ k ∈ proc_keys σ. unat (s (@proc_id k)) = proc_id σ k"
```

A storage corresponds to the abstract state if and only if the above properties are satisfied.

```
definition models :: "storage ⇒ ('p :: proc_class) abs ⇒ bool" ("_ ⊩ _" [65, 65] 65) where
  "s ⊩ σ ≡
    models_nprocs s σ
  ∧ models_proc_keys s σ
  ∧ models_proc_ids s σ"

lemmas models_nprocs = models_def models_nprocs_def
lemmas models_proc_keys = models_def models_proc_keys_def
lemmas models_proc_ids = models_def models_proc_ids_def
```

In the following we aim to proof the existence of a storage corresponding to any well-formed abstract state (so that any well-formed abstract state can be encoded and stored). Then prove that the encoding is unambiguous.

5.1 Auxiliary definitions and lemmas

An empty storage:

```
definition "zero_storage (_ :: word32) ≡ 0 :: word32"
```

The set of all procedure key addresses:

```
definition proc_key_addrs ("@proc'_keys") where
  "@proc_keys σ ≡ { @proc_key (proc_id σ k) | k. k ∈ proc_keys σ }"

definition proc_id_addrs ("@proc'_ids") where "@proc_ids σ ≡ { @proc_id k | k. k ∈ proc_keys σ }"
```

Procedure id can be converted to a 32-byte word without overflow.

```
lemma proc_id_inv[simp]:
  "⟦⊢ σ; k ∈ proc_keys σ⟧ ⟹ unat (of_nat (proc_id σ k) :: word32) = proc_id σ k"
  unfolding procs_rng_wf
  by (force intro:le_unat_uoi[where z=max_nprocs_word])
```

Moreover, any procedure id is non-zero and bounded by the maximum available id (*max_nprocs_word*).

```
lemma proc_id_bounded[intro]:
  "⟦⊢ σ; k ∈ proc_keys σ⟧ ⟹
   (0 :: word32) < of_nat (proc_id σ k) ∧ of_nat (proc_id σ k) ≤ max_nprocs_word"
  by (simp add:word_le_nat_alt word_less_nat_alt, force simp add:procs_rng_wf)
```

Since it's non-zero, any procedure id has a non-zero bit in its lower part.

```
lemma proc_id_low_one:
  "0 < n ∧ n ≤ max_nprocs_word ⟹ ∃i∈{0..<proc_id_len}. n !! i"
  (is "?nbound ⟹ _")
proof–
  assume 0:"?nbound"
  hence "¬ ?thesis ⟹ n = 0" by (auto simp add:inc_le intro!:word_eqI)
  moreover from 0 have "n ≠ 0" by auto
  ultimately show ?thesis by auto
qed
```

And procedure key address is different from the address of the # of procedures (*@nprocs*).

```
lemma proc_key_addr_neq_nprocs_key:
  "0 < n ∧ n ≤ max_nprocs_word ⟹ @nprocs OR n ≠ @nprocs"
  (is "?nbound ⟹ _")
proof–
  assume 0:"?nbound"
  hence "∃ i∈{0..<proc_id_len}.(@nprocs !! i ∨ n !! i) ≠ @nprocs !! i"
    using nprocs_key_low_zeros proc_id_low_one
    by fast
  thus ?thesis by (force simp add:word_eq_iff word_ao_nth)
qed
```

Thus *@nprocs* doesn't belong to the set of procedure key addresses.

```
lemma nprocs_key_notin_proc_key_addrs: "⊢ σ ⟹ @nprocs ∉ @proc_keys σ"
  using proc_id_bounded proc_key_addr_neq_nprocs_key
  unfolding proc_key_addrs_def proc_key_addr_def
  by auto
```

Also procedure index address is different from the address of the # of procedures (*@nprocs*).

```
lemma proc_id_addr_neq_nprocs_key: "@proc_id k ≠ @nprocs"
proof
  have 0: "¬ @nprocs !! 0" by auto
  have 1: "@proc_id k !! 0" using lsb0 test_bit_1 by blast
  assume "@proc_id k = @nprocs"
  hence "(@proc_id k !! 0) = (@nprocs !! 0)" by auto
  thus "False" using 0 1 by auto
qed
```

Thus *@nprocs* doesn't belong to the set of procedure index addresses.

```
lemma nprocs_key_notin_proc_id_addrs: "⊢ σ ⟹ @nprocs ∉ @proc_ids σ"
  unfolding proc_id_addrs_def
proof
  assume assms: "⊢ σ" and "@nprocs ∈ {@proc_id k |k. k ∈ proc_keys σ}"
  hence "∃k. @nprocs = @proc_id k ∧ k ∈ proc_keys σ" by blast
  then obtain k where " @nprocs = @proc_id k ∧ k ∈ proc_keys σ" by blast
```

```
  thus "False" using proc_id_addr_neq_nprocs_key assms by auto
qed
```

The function mapping procedure id to the corresponding procedure key (in some abstract state):

```
definition "proc_key_of_id σ ≡ the_inv_into (proc_keys σ) (proc_id σ)"
```

Invertibility of computing procedure id (by its key) in any abstract state:

```
lemma proc_key_of_id_inv[simp]: "⟦⊢σ; k ∈ proc_keys σ⟧ ⟹ proc_key_of_id σ (proc_id σ k) = k"
  unfolding procs_map_wf proc_key_of_id_def
  using the_inv_into_f_f by fastforce
```

For any valid procedure id in any well-formed abstract state there is a procedure key that corresponds to the id (this is not so trivial as we keep the reverse mapping in the abstract state, the proof is implicitly based on the pigeonhole principle).

```
lemma proc_key_exists: "⟦⊢σ; i∈{1..nprocs σ}⟧ ⟹ ∃k∈proc_keys σ. proc_id σ k = i"
proof (rule ccontr, subst (asm) bex_simps(8))
  let ?rng = "{1 .. nprocs σ}"
  let ?prj = "proc_id σ ` proc_keys σ"
  assume "∀k∈proc_keys σ. proc_id σ k ≠ i"
  hence 0:"i ∉ ?prj"
    by auto
  assume "⊢ σ"
  hence 1:"?prj ⊆ ?rng" and 2:"card ?prj = card ?rng"
    unfolding abs_wf_def procs_rng_wf_def procs_map_wf_def
    by (auto simp add: image_subset_iff card_image)
  assume *:"i∈?rng"
  have "card ?prj = card (?prj ∪ {i} − {i})"
    using 0 by simp
  also have "... < card (?prj ∪ {i})"
    by (rule card_Diff1_less, simp_all)
  also from * have "... ≤ card ?prj"
    using 1
    by (subst 2, intro card_mono, simp_all)
  finally show False ..
qed
```

The function *proc_key_of_id* gives valid procedure ids.

```
lemma proc_key_of_id_in_keys[simp]: "⟦⊢σ; i∈{1..nprocs σ}⟧ ⟹ proc_key_of_id σ i ∈ proc_keys σ"
  using proc_key_exists the_inv_into_into[of "proc_id σ" "proc_keys σ" i]
  unfolding proc_key_of_id_def procs_map_wf
  by fast
```

Invertibility of computing procedure key (by its id) in any abstract state:

```
lemma proc_key_of_id_inv'[simp]: "⟦⊢σ; i∈{1..nprocs σ}⟧ ⟹ proc_id σ (proc_key_of_id σ i) = i"
  using proc_key_exists f_the_inv_into_f[of "proc_id σ" "proc_keys σ" i]
  unfolding proc_key_of_id_def procs_map_wf
  by fast
```

5.2 Any well-formed abstract state can be stored

A mapping of addresses with specified (defined) values:

```
definition
  "con_wit_map σ :: _ ⇀ word32 ≡
      [@nprocs ↦ of_nat (nprocs σ)]
    ++ (Some ∘ ucast ∘ proc_key_of_id σ ∘ id_of_proc_key_addr) |` @proc_keys σ
    ++ (Some ∘ ucast ∘ proc_key_of_id_addr) |` @proc_ids σ"
```

A sample storage extending the above mapping with default zero values:

```
definition "con_wit σ ≡ override_on zero_storage (the ∘ con_wit_map σ) (dom (con_wit_map σ))"

lemmas con_wit = con_wit_def con_wit_map_def comp_def

lemma restrict_subst[simp]: "k ∈ s ⟹ (f |` { g k | k. k ∈ s}) (g k) = f (g k)"
  unfolding restrict_map_def
  by auto

lemma restrict_rule: "x ∉ A ⟹ x ∉ dom(f |` A)"
  by simp
```

Existence of a storage corresponding to any well-formed abstract state:

```
theorem models_nonvac: "⊢ σ ⟹ ∃s. s ⊩ σ"
  unfolding models_nprocs models_proc_keys models_proc_ids
proof (intro exI[of _ "con_wit σ"] conjI)
  assume wf:"⊢σ"
  thus "unat (con_wit σ @nprocs) = nprocs σ"
    unfolding con_wit
   using nprocs_key_notin_proc_key_addrs nprocs_key_notin_proc_id_addrs le_unat_uoi[where z=max_nprocs_word]
    apply (subst override_on_apply_in, simp, subst map_add_dom_app_simps(3))
    apply (rule restrict_rule, auto, subst map_add_dom_app_simps(3))
    by (auto simp add:procs_rng_wf)
  from wf have "⋀ k. k ∈ proc_keys σ ⟹
                  proc_key_of_id σ (id_of_proc_key_addr (@proc_key (proc_id σ k))) = k"
    by (subst id_of_key_addr_inv) (auto simp add:procs_rng_wf, force)
  thus "∀k∈proc_keys σ. con_wit σ (@proc_key (proc_id σ k)) = ucast k"
    unfolding con_wit proc_key_addrs_def proc_id_addrs_def
    apply (intro ballI, subst override_on_apply_in, (auto)[1])
    apply (subst map_add_dom_app_simps(3))

    sorry
  show "∀k∈proc_keys σ. unat (con_wit σ (@proc_id k)) = proc_id σ k"
    unfolding con_wit proc_key_addrs_def proc_id_addrs_def
    sorry
qed
```

5.3 Unambiguity of encoding

5.3.1 Auxiliary lemmas

```
proposition word32_key_downcast:"is_down (ucast :: word32 ⇒ key)"
  unfolding is_down_def target_size_def source_size_def
  by simp

lemmas key_upcast =
  ucast_down_ucast_id[OF word32_key_downcast]
  down_ucast_inj[OF word32_key_downcast]

lemma con_id_inj[consumes 4]:
  "⟦⊢ σ; s ⊩ σ;
    i₁∈{1..nprocs σ}; i₂∈{1..nprocs σ};
    s (@proc_key i₁) = s (@proc_key i₂)⟧ ⟹ i₁ = i₂"
  unfolding models_proc_keys
  using proc_key_of_id_in_keys key_upcast
      proc_key_of_id_inv'[symmetric, of _ i₁] proc_key_of_id_inv'[symmetric, of _ i₂]
  by metis
```

The concrete encoding of abstract storage is unambiguous, i. e. the same storage cannot model two distinct well-formed abstract states.

```
theorem models_inj[simp]: "⟦⊢ σ₁; ⊢ σ₂; s ⊩ σ₁; s ⊩ σ₂⟧ ⟹ (σ₁ :: ('p :: proc_class) abs) = σ₂"
```

```
  (is "⟦?wf1; ?wf2; ?models1; ?models2⟧ ⟹ _")
proof (intro abs.equality ext option.expand, rule ccontr)
  fix x
  {
    fix σ σ'::"'p abs"
    assume wf1:"⊩ σ" and wf2:"⊩ σ'"
    assume models1:"s ⊩ σ" and models2:"s ⊩ σ'"
    fix i p
    assume Some:"procs σ x = Some (i, p)"
    with wf1 have "i∈{1..nprocs σ}" unfolding procs_rng_wf Ball_def by auto
    with wf2 models1 models2
    have "proc_key_of_id σ' i ∈ proc_keys σ' ∧ proc_id σ' (proc_key_of_id σ' i) = i"
      unfolding models_nprocs by simp
    moreover with Some models1 models2 have "proc_key_of_id σ' i = x"
      unfolding models_proc_keys
      using key_upcast by (metis domI fst_conv option.sel)
    ultimately have "procs σ' x ≠ None" by auto
  }
  note wlog = this
  assume ?wf1 ?wf2 ?models1 and ?models2
  {
    assume neq:"(procs σ₁ x = None) ≠ (procs σ₂ x = None)"
    show False
    proof (cases "procs σ₁ x")
      case Some
      with wlog ‹?wf1› ‹?wf2› ‹?models1› ‹?models2› neq show ?thesis by fastforce
    next
      case None
      with neq have "procs σ₂ x ≠ None" by simp
      with wlog ‹?wf1› ‹?wf2› ‹?models1› ‹?models2› neq show ?thesis by force
    qed
  }
  {
    assume inσ₁:"procs σ₁ x ≠ None" and inσ₂:"procs σ₂ x ≠ None"
    show "proc σ₁ x = proc σ₂ x"
    proof
      let ?i₁ = "proc_id σ₁ x" and ?i₂ = "proc_id σ₂ x"
      from inσ₁ and inσ₂
      have "procs σ₁ x = Some (?i₁, proc_bdy σ₁ x)"
        and "procs σ₂ x = Some (?i₂, proc_bdy σ₂ x)"
        by auto
      with ‹?wf1› and ‹?wf2›
      have "?i₁∈{1..nprocs σ₁}" and "?i₂∈{1..nprocs σ₂}" unfolding procs_rng_wf by auto
      moreover with inσ₁ inσ₂ ‹?models1› and ‹?models2›
      have "s (@proc_key ?i₁) = s (@proc_key ?i₂)" unfolding models_proc_keys by force
      moreover with ‹?models1› ‹?models2› and ‹?i₂∈{1..nprocs σ₂}›
      have "?i₂∈{1..nprocs σ₁}" unfolding models_nprocs by simp
      ultimately show "?i₁ = ?i₂" using ‹?wf1› ‹?models1› and con_id_inj[of σ₁] by blast

      show "proc_bdy σ₁ x = proc_bdy σ₂ x"
        using inσ₁ inσ₂ ‹?wf1› ‹?wf2› key_inj
        unfolding procs_rng_wf by (metis UNIV_I domIff the_inv_into_f_f)
    qed
  }
qed (simp)
```

6 Well-formedness of a storage state

We need a decoding function on storage states. However, not every storage state can be decoded into an abstract state. So we introduce a minimal well-formedness predicate on storage states.

Number of procedures is bounded, otherwise procedure key addresses can become invalid and we cannot read the procedure keys from the storage.

definition "$nprocs\_wf\ s \equiv unat\ (s\ @nprocs) \leq max\_nprocs$"

Well-formedness of procedure keys:

1. procedure keys should fit into 24-byte words;
2. they should represent some existing procedures (currently this is essentially a temporary work-around and is understood in a very abstract sense (Hilbert epsilon operator is used to "retrieve" procedures), really we need to formalize how the procedures themselves are stored);
3. the same procedure key should not be stored by two distinct procedure key addresses;
4. procedure heap should contain valid procedure index for each procedure key.

definition "$proc\_keys\_wf\ (dummy :: 'a\ itself)\ (s :: storage) \equiv$
$(\forall k \in \{s\ (@proc\_key\ i) \mid i.\ i \in \{1..unat\ (s\ @nprocs)\}\}.\ ucast\ (ucast\ k :: key) = k)$
$\wedge\ (\forall i \in \{1..unat\ (s\ @nprocs)\}.\ \exists p :: ('a :: proc\_class).\ ucast\ (key\ p) = s\ (@proc\_key\ i))$
$\wedge\ inj\_on\ (s \circ @proc\_key)\ \{1..unat\ (s\ @nprocs)\}$
$\wedge\ (\forall i \in \{1\ ..\ unat\ (s\ @nprocs)\}.\ unat\ (s\ (@proc\_id\ (ucast\ (s\ (@proc\_key\ i))))) = i)$"

Well-formedness of a storage state: the two above requirements should hold.

definition con_wf ("$\models_{\_}\ \_$" [1000, 60] 60) **where**
"$\models_{(d :: ('a :: proc\_class)\ itself)}\ s \equiv$
$nprocs\_wf\ s$
$\wedge\ proc\_keys\_wf\ d\ s$"

notation (*input*) con_wf ("$\models_{\_}$" [1000, 60] 60)

lemmas $nprocs\_wf = con\_wf\_def\ nprocs\_wf\_def$

lemmas $proc\_keys\_wf = con\_wf\_def\ proc\_keys\_wf\_def$

We proceed with the proof that any storage state corresponding (in the $\Vdash$ sense) to a well-formed abstract state is well-formed.

6.1 Auxiliary lemmas

Any property on procedure ids can be reformulated on the corresponding procedure keys according to a well-formed abstract state (elimination rule for procedure ids).

lemma *elim_proc_id*[*consumes 3*]:
assumes "$i \in \{1..unat\ (s\ @nprocs)\}$"
assumes "$\vdash \sigma$"
assumes "$s \Vdash \sigma$"
obtains k **where** "$k \in proc\_keys\ \sigma \wedge i = proc\_id\ \sigma\ k$"
using *assms proc_key_exists*
unfolding *models_nprocs*
by *metis*

6.2 Storage corresponding to a well-formed state is well-formed

theorem *model_wf*[*simp, intro*]:"$[\![\vdash (\sigma :: ('p :: proc\_class)\ abs);\ s \Vdash \sigma]\!] \Longrightarrow \models_{(p :: 'p\ itself)} s$"
unfolding *proc_keys_wf*
proof (*intro conjI ballI*)

```
  assume wf:"⊢ σ" and models:"s ⊩ σ"
  thus "nprocs_wf s"
    unfolding procs_rng_wf models_nprocs nprocs_wf_def by simp
  note elim_id = elim_proc_id[OF _ wf models]
  show "inj_on (s ∘ @proc_key) {1..unat (s @nprocs)}"
    unfolding inj_on_def
  proof (intro ballI impI)
    fix x y
    assume "x∈{1..unat (s @nprocs)}" and "y∈{1..unat (s @nprocs)}"
    from wf models and this
    show "(s ∘ @proc_key) x = (s ∘ @proc_key) y ⟹ x = y"
      using key_upcast
      by (elim elim_id, auto simp add:models_proc_keys)
  qed
  {
    fix i
    assume "i∈{1..unat (s @nprocs)}"

    thus "∃ p :: 'p. ucast (key p) = s (@proc_key i)"
      using wf models
      apply (intro exI[of _ "proc_bdy σ (proc_key_of_id σ i)"])
      by (elim elim_id, simp add:models_proc_keys procs_rng_wf)
  }
  {
    fix k
    assume "k∈{s (@proc_key i) | i. i∈{1..unat (s @nprocs)}}"
    then obtain x where "x ∈ {1..unat (s @nprocs)}" and "k = s (@proc_key x)"
      by (simp only:Setcompr_eq_image image_iff, elim bexE)
    thus "ucast (ucast k :: key) = k"
      using wf models key_upcast
      by (elim elim_id, auto simp add:models_proc_keys)
  }
  fix i
  assume "i ∈ {1..unat (s @nprocs)}"
  thus "unat (s (@proc_id (ucast (s (@proc_key i))))) = i"
    using wf models
    sorry
qed
```

7 Decoding of storage

Auxiliary abbreviations

```
abbreviation "proc_pair (p :: ('p :: proc_class) itself) s i ≡
  let k = ucast (s (@proc_key i)) in (k, (i, SOME p :: 'p. key p = k))"

abbreviation "proc_list p s ≡ [proc_pair p s i. i ← [1..<Suc (unat (s @nprocs))]]"
```

The decoding function:

```
definition abs ("⦃_⦄_" [1000, 1000] 1000) where "⦃s⦄p = ⦇ procs = map_of (proc_list p s) ⦈"

notation (input) abs ("⦃_⦄_" [1000, 1000] 1000)

lemmas abs_simps =
  Let_def set_map image_iff set_upt atLeastLessThanSuc_atLeastAtMost
  abs.simps option.sel fst_conv

theorem models_abs[simp, intro]: "⊫p s ⟹ s ⊢ ⦃s⦄p"
  unfolding models_nprocs models_proc_keys models_proc_ids
proof (intro conjI ballI)
```

```
  assume wf:"⊫p s"
  hence "inj_on (λi. ucast (s (@proc_key i)) :: key) {1..unat (s @nprocs)}"
    unfolding inj_on_def proc_keys_wf
    by (auto simp only:comp_apply Ball_def mem_Collect_eq) metis
  hence fst_inj:"inj_on fst (set (proc_list p s))"
    unfolding inj_on_def set_upt Ball_def abs_simps
    by simp
  have dist:"distinct (proc_list p s)"
    by (simp add:distinct_map inj_on_def Let_def)
  show models_nprocs:"unat (s @nprocs) = nprocs ⦃s⦄p"
    unfolding abs_def
    by (simp only:abs.simps dom_map_of_conv_image_fst
        distinct_card[OF dist] card_image[OF fst_inj] length_map length_upt)

  have proc_pair_inj:"inj_on (proc_pair p s) {1..unat (s @nprocs)}"
    unfolding inj_on_def prod.inject Let_def by simp
  fix k
  {
    fix i q
    assume proc:"procs ⦃s⦄p k = Some (i, q)"
    hence "(k, proc ⦃s⦄p k) = proc_pair p s i"
      by (simp only:abs.simps abs_def) (frule map_of_SomeD, auto simp add:Let_def)
  } note proc_k_eq = this
  {
  assume k_in_keys:"k∈proc_keys ⦃s⦄p"
  hence proc_id_in_range:"proc_id ⦃s⦄p k ∈ {1..nprocs ⦃s⦄p}"
    apply (subst models_nprocs[symmetric])
    unfolding abs_def by (auto simp only:abs_simps; frule map_of_SomeD)+
  have "map_of (proc_list p s) k = Some (proc ⦃s⦄p k)"
    using fst_inj proc_pair_inj proc_k_eq k_in_keys
    apply (auto simp only:distinct_map distinct_upt abs_simps intro!:map_of_is_SomeI)
    using models_nprocs proc_id_in_range
    by (intro bexI[of _ "proc_id ⦃s⦄p k"], simp+)
  thus "s (@proc_key (proc_id ⦃s⦄p k)) = ucast k"
    unfolding abs_def
    apply (cases "map_of (proc_list p s) k")
    apply (auto simp only: abs_simps, frule map_of_SomeD)
    using wf unfolding proc_keys_wf by (force simp only: abs_simps)
  }
  next
  fix k
  assume "k ∈ proc_keys ⦃s⦄p"
  thus "unat (s (@proc_id k)) = proc_id ⦃s⦄p k"
    sorry
qed
```

8 System calls

This section will contain specifications of the system calls.

```
locale syscall =
  fixes arg_wf :: "'p :: proc_class abs ⇒ 'b ⇒ bool" ("_ ⊢ _" [60, 60] 60)
  fixes arg_abs :: "'a ⇒ 'b" ("⦃_⦄")
  fixes pre :: "'a ⇒ storage ⇒ bool"
  fixes post :: "'a ⇒ storage ⇒ storage ⇒ bool"
  fixes app :: "'b ⇒ 'p abs ⇒ 'p abs"
  assumes preserves_wf: "⟦⊢σ; σ ⊢ arg⟧ ⟹ ⊢ app arg σ"
  assumes preserves_wf': "⟦⊫p s; ⊢⦃s⦄p; pre a s; post a s s'⟧ ⟹ ⊫p s'"
  assumes arg_wf: "⟦⊫p s; ⊢⦃s⦄p; pre a s⟧ ⟹ ⦃s⦄p ⊢ ⦃a⦄"
  assumes consistent: "⟦⊫p s; ⊢⦃s⦄p; pre a s; post a s s'⟧ ⟹ ⦃s'⦄p = app ⦃a⦄ ⦃s⦄p"
```

```
begin
theorem post_wf: "⟦⊫p s; ⊢(⦃s⦄p :: 'p abs); pre a s; post a s s'⟧ ⟹ ⊢⦃s'⦄p"
  using arg_wf preserves_wf consistent by metis
end

definition add_proc_arg_wf :: "'p :: proc_class abs ⇒ (key × 'p) ⇒ bool" ("_ ⊢add'_proc _")
  where
  "σ ⊢add_proc kp ≡
    let (k, p) = kp in
    nprocs σ < max_nprocs ∧
    k ∉ proc_keys σ ∧
    key p = k"

definition "add_proc kp σ ≡ σ ⦇ procs := procs σ (fst kp ↦ (nprocs σ + 1, snd kp)) ⦈"

definition add_proc_arg_abs :: "(key × 'p :: proc_class) ⇒ (key × 'p)" ("⦃_⦄add'_proc") where
  "⦃a⦄add_proc = a"

definition
  "add_proc_pre (kp :: _ × 'p :: proc_class) s ≡ ⦃s⦄TYPE('p) ⊢add_proc ⦃kp⦄add_proc"

definition
  "add_proc_post kp s s' ≡
    let (k, p) = kp in
    s' @nprocs = s @nprocs + 1 ∧
    (∀ a∈{ @proc_key i | i. i∈{1..unat (s @nprocs)}}. s' a = s a) ∧
    s' (@proc_key (unat (s @nprocs) + 1)) = k"

lemma add_proc_preserves_wf: "⟦⊢σ; σ ⊢add_proc (k, p)⟧ ⟹ ⊢ add_proc (k, p) σ"
  (is "⟦?wfσ; ?wf_arg⟧ ⟹ _")
proof (subst abs_wf_def, unfold procs_rng_wf_def procs_map_wf_def, intro conjI ballI)
  let ?σ' = "add_proc (k, p) σ"
  assume ?wfσ and ?wf_arg

  thus "nprocs (?σ') ≤ max_nprocs" unfolding add_proc_arg_wf_def add_proc_def by simp

  have proc_keys':"proc_keys ?σ' = proc_keys σ ∪ {k}" unfolding add_proc_def by simp
  have proc_id_k:"proc_id ?σ' k = nprocs σ + 1" unfolding add_proc_def by simp
  have proc_id_unch:"∀ k∈proc_keys σ. proc_id ?σ' k = proc_id σ k"
    using ‹?wf_arg› unfolding add_proc_def add_proc_arg_wf_def by simp

  show "inj_on (proc_id ?σ') (proc_keys ?σ')"
  proof (unfold inj_on_def, intro ballI impI)
    fix x y
    assume "x ∈ proc_keys ?σ'" and "y ∈ proc_keys ?σ'" and "proc_id ?σ' x = proc_id ?σ' y"
    with ‹?wfσ› proc_keys' proc_id_k proc_id_unch show "x = y"
      unfolding procs_map_wf procs_rng_wf inj_on_def add_proc_arg_wf_def
           Ball_def atLeastAtMost_iff
      by (cases "x ∈ proc_keys σ"; cases "y ∈ proc_keys σ", auto)
  qed

  fix k'
  assume k'_in_keys:"k'∈proc_keys ?σ'"

  with ‹?wfσ› ‹?wf_arg› proc_keys' proc_id_k proc_id_unch
  show "proc_id ?σ' k' ∈ {1..nprocs ?σ'}"
    unfolding add_proc_arg_wf_def procs_rng_wf Ball_def atLeastAtMost_iff
    by (cases "k' = k", auto)

  have "proc_bdy ?σ' k = p" unfolding add_proc_def by simp
```

```
  moreover have "∀k∈proc_keys σ. proc_bdy ?σ' k = proc_bdy σ k"
    using ‹?wf_arg› unfolding add_proc_def add_proc_arg_wf_def by simp
  ultimately show "key (proc_bdy ?σ' k') = k'"
    using k'_in_keys ‹?wf σ› ‹?wf_arg› proc_keys'
    unfolding add_proc_arg_wf_def procs_rng_wf
    by (cases "k' = k", auto)
qed
```

8.1 Register Procedure

Early version of "register procedure" operation.

```
abbreviation higher32 where "higher32 k n ≡ n >> ((32 − k) ∗ 8)"

definition is_kernel_storage_key :: "word32 ⇒ bool"
  where "is_kernel_storage_key w ≡ higher32 4 w = 0xffffffff"


definition n_of_procedures :: "storage ⇒ word32"
  where "n_of_procedures s = s @nprocs"


definition add_proc' :: "key ⇒ storage ⇒ storage option"
  where
    "add_proc' p s ≡
     if n_of_procedures s < max_nprocs_word
     then
       Some (s
            (@nprocs := n_of_procedures s + 1,
             @nprocs OR (n_of_procedures s + 1) := ucast p))
     else
       None"

lemma
  assumes "n_of_procedures s < max_nprocs_word"
  shows   "case (add_proc' p s) of
           Some s' ⇒ n_of_procedures s' = n_of_procedures s + 1"
proof−
  have "0 < n_of_procedures s + 1"
    using assms
    by (metis
          add_cancel_right_right
          inc_i word_le_0_iff word_le_sub1 word_neq_0_conv word_zero_le zero_neq_one)
  moreover have "n_of_procedures s + 1 ≤ max_nprocs_word"
    using assms
    by (metis add.commute add.right_neutral inc_i word_le_sub1 word_not_simps(1) word_zero_le)
  ultimately have "@nprocs OR n_of_procedures s + 1 ≠ @nprocs"
    using proc_key_addr_neq_nprocs_key by auto
  thus ?thesis
    unfolding add_proc'_def
    using assms
    by (simp add:n_of_procedures_def)
qed
end
```