# Formal specification of the Cap9 kernel

Mikhail Mandrykin     Ilya Shchepetkov

May 8, 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 state.

**type-synonym** *word24* = *192 word*
**type-synonym** *key* = *word24*

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.

We accomplish it by using Isar type classes:

**class** *proc-class* =
  **fixes** *key* :: $'a \Rightarrow$ *key*
  **assumes** $CARD\ ('a) \leq CARD\ (key)$
  **assumes** *inj key*

To insure we don't introduce contradictions with these assumptions we build a sample model of the set of all procedures. To do this we proceed with some additional definitions.

Byte is 8-bit machine word:

**type-synonym** *byte* = *8 word*

## 2.2 Hash function

This is a hash function that takes an arbitrary list of bytes and returns its 24-byte hash. It is used to obtain procedure keys.

```
fun hash-rec :: nat ⇒ byte list ⇒ key where
  hash-rec n []              = 0 |
  hash-rec 0 [e]             = ucast e << 191 |
  hash-rec (Suc n) [e]       = ucast e << n |
  hash-rec 0 (e # es)        = (ucast e << 191) XOR hash-rec 191 es |
  hash-rec (Suc n) (e # es) = (ucast e << n) XOR hash-rec n es
```

```
definition some-hash ≡ hash-rec 0
```

This is an auxiliary function that takes some hash as an input and if there is some byte list (called "element"), whose hash matches the input, then the function will return it. Otherwise, the function returns an empty set:

```
definition choose-proc k ≡
  if k = 0                       then {[]}
  else if ∃ p. some-hash p = k then {SOME p. some-hash p = k}
  else                              {}
```

```
lemma choose-proc[simp]: x ∈ choose-proc k ⟹ some-hash x = k
  unfolding choose-proc-def
  by (auto simp add: some-hash-def split: if-splits intro: someI)
```

Each key has only one corresponding procedure:

```
lemma[simp]: ⟦x ∈ choose-proc k; y ∈ choose-proc k⟧ ⟹ x = y
  unfolding choose-proc-def
  by (simp split: if-splits)
```

## 2.3 Procedures

*Procs* is a set of all possible procedures, hash of which will be a valid procedure key:

```
definition Procs ≡ ⋃k. choose-proc k
```

Here we introduce a new type called *proc* which will be used to represent procedures in the abstract state. Type *proc* is identified with the *Procs* set:

```
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 some-hash-inj: inj-on some-hash Procs
```

**unfolding** *inj-on-def Procs-def*
**by** *auto*

### 2.3.2 Number of all procedures

Here we introduce maximum number of procedure keys:

**abbreviation** *max-nkeys* $\equiv$ *2 ^ 192* :: *nat*

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

**lemma** *card-procs*: *card Procs* $\leq$ *max-nkeys*
**unfolding** *Procs-def*
**proof** (*subst card-UN-disjoint*)
**show** *finite* (*UNIV* :: *key set*)
**and** $\forall$ *i*$\in$*UNIV*. *finite* (*choose-proc i*)
**unfolding** *choose-proc-def some-hash-def*
**by** (*simp-all split*: *if-splits*)
**show** $\forall$ *i*$\in$*UNIV*. $\forall$ *j*$\in$*UNIV*. *i* $\neq$ *j* $\longrightarrow$ *choose-proc i* $\cap$ *choose-proc j* = {}
**by** (*auto*, ((*drule choose-proc*)+, *simp*))
**show** ($\sum$ *i*$\in$*UNIV*. *card* (*choose-proc i*)) $\leq$ *max-nkeys*
**using** *sum-bounded-above*[*of UNIV* :: *key set* $\lambda$ *i*. *card* (*choose-proc i*), **where** *K* = *1*]
**unfolding** *choose-proc-def card-word*
**by** *auto*
**qed**

## 2.4 Sample set of procedures

Here we show that there is a sample set of all procedures that satisfies all assumptions:

**instantiation** *proc* :: *proc-class*
**begin**
**definition** *key* $\equiv$ *some-hash* $\circ$ *Rep-proc*

**instance proof**
**show** *CARD*(*proc*) $\leq$ *CARD*(*key*)
**using** *card-procs*
*card-word*[**where** *'a* = *192*]
*type-definition.card*[*OF proc.type-definition-proc*]
**by** *auto*
**show** *inj* (*key* :: *proc* $\Rightarrow$ -)
**using** *some-hash-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* $\rightharpoonup$ *nat* $\times$ $'p$

## 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* $\mathcal{S} \equiv$ *card* (*dom* (*procs* $\mathcal{S}$))

List of procedures keys:

**abbreviation** *proc-keys* $\mathcal{S} \equiv$ *dom* (*procs* $\mathcal{S}$)

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

**abbreviation** *proc* $\mathcal{S}$ $k \equiv$ *the* (*procs* $\mathcal{S}$ $k$)

Procedure index for a given key:

**abbreviation** *proc-id* $\mathcal{S}$ $k \equiv$ *fst* (*proc* $\mathcal{S}$ $k$)

Procedure itself for a given key:

**abbreviation** *proc-bdy* $\mathcal{S}$ $k \equiv$ *snd* (*proc* $\mathcal{S}$ $k$)

Maximum number of procedures in the abstract state:

**abbreviation** *max-nprocs-nat* $\equiv$ *2* ^ *24* $-$ *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* $\mathcal{S} \equiv$
  ($\forall$ $k \in$ *proc-keys* $\mathcal{S}$. *proc-id* $\mathcal{S}$ $k \in \{1\ ..\ nprocs\ \mathcal{S}\}$ $\wedge$ *key* (*proc-bdy* $\mathcal{S}$ $k$) $= k$) $\wedge$
  *nprocs* $\mathcal{S} \leq$ *max-nprocs-nat*

Procedure indexes must be injective:

**definition** *procs-map-wf* $\mathcal{S} \equiv$ *inj-on* (*proc-id* $\mathcal{S}$) (*proc-keys* $\mathcal{S}$)

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

**definition** *abs-wf* :: *'p* :: *proc-class abs* $\Rightarrow$ *bool* ($\vdash$- [*60*]) **where**
$\vdash\mathcal{S} \equiv$
*procs-rng-wf* $\mathcal{S}$
$\wedge$ *procs-map-wf* $\mathcal{S}$

**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* $\Rightarrow$ *word32*

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

**abbreviation** *nprocs-key* $\equiv$ *0xffffffff01* << (*27* $*$ *8*) :: *word32*

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

**abbreviation** *key-addr-of-id i* $\equiv$ *nprocs-key OR of-nat i*

Procedure index that corresponds to some procedure key address:

**abbreviation** *id-of-key-addr a* $\equiv$ *unat* (*nprocs-key XOR a*)

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

**abbreviation** *max-nprocs-word* $\equiv$ *2* ^ *24* − *1* :: *word32*

Declare some lemmas as simplification rules:

**declare** *unat-word-ariths*[*simp*] *word-size*[*simp*]

## 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*∈{*24..<256*}. ¬ *n* !! *i* (**is** *?nbound* ⟹ ∀ - ∈ *?high*. -)
**proof**
  **fix** *i*
  **assume** *0*:*i* ∈ *?high*
  **from** *0* **have** *2* ^ *24* ≤ (*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* ^ *24* ≤ (*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*) (*frule 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..< 24*}. ¬ *nprocs-key* !! *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..<24*}. *P* (*x* !! *n*)) ∧ (∀ *n*∈ {*24..<256*}. *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-nat* ⟹ *id-of-key-addr* (*key-addr-of-id 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-key XOR nprocs-key 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*

**by** *simp*
**qed**

# 5 Correspondence between abstract and storage states

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

**definition** *models-nprocs* $S\ \mathcal{S} \equiv$ *unat* (*S nprocs-key*) = *nprocs* $\mathcal{S}$

Each procedure key $k$ is stored by the corresponding address (*key-addr-of-id* $k$).

**definition** *models-proc-keys* $S\ \mathcal{S} \equiv$
$\forall\ k \in$ *proc-keys* $\mathcal{S}$. $S$ (*key-addr-of-id* (*proc-id* $\mathcal{S}$ $k$)) = *ucast* $k$

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

**definition** *models* :: *storage* $\Rightarrow$ ($'p$ :: *proc-class*) *abs* $\Rightarrow$ *bool* (- $\Vdash$ - [*65*]) **where**
$S \Vdash \mathcal{S} \equiv$
*models-nprocs* $S\ \mathcal{S}$
$\wedge$ *models-proc-keys* $S\ \mathcal{S}$

**lemmas** *models-nprocs* = *models-def models-nprocs-def*

**lemmas** *models-proc-keys* = *models-def models-proc-keys-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). Later we still need to prove that the encoding is unambiguous.

## 5.1 Auxiliary definitions and lemmas

An empty storage.

**definition** *zero-con* (- :: *word32*) $\equiv$ *0* :: *word32*

The set of all procedure key addresses.

**definition** *proc-key-addrs* $\mathcal{S} \equiv \{$ *key-addr-of-id* (*proc-id* $\mathcal{S}$ $k$) | $k$. $k \in$ *proc-keys* $\mathcal{S}$ $\}$

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

**lemma** *proc-id-inv*[*simp*]:
$[\![\vdash \mathcal{S};\ k \in$ *proc-keys* $\mathcal{S}]\!] \Longrightarrow$ *unat* (*of-nat* (*proc-id* $\mathcal{S}$ $k$) :: *word32*) = *proc-id* $\mathcal{S}$ $k$
**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*]:
$[\![\vdash \mathcal{S};\ k \in \mathit{proc\text{-}keys}\ \mathcal{S}]\!] \Longrightarrow$
$(0 :: \mathit{word32}) < \mathit{of\text{-}nat}\ (\mathit{proc\text{-}id}\ \mathcal{S}\ k) \wedge \mathit{of\text{-}nat}\ (\mathit{proc\text{-}id}\ \mathcal{S}\ k) \leq \mathit{max\text{-}nprocs\text{-}word}$
**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*:
$0 < n \wedge n \leq \mathit{max\text{-}nprocs\text{-}word} \Longrightarrow \exists\, i{\in}\{0..{<}24\}.\ n\ !!\ i$ (**is** *?nbound* $\Longrightarrow$ -)
**proof**−
**assume** *0*:*?nbound*
**hence** $\neg$ *?thesis* $\Longrightarrow n = 0$ **by** (*auto simp add*:*inc-le intro*!:*word-eqI*)
**moreover from** *0* **have** $n \neq 0$ **by** *auto*
**ultimately show** *?thesis* **by** *auto*
**qed**

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

**lemma** *proc-key-addr-neq-nprocs-key*:
$0 < n \wedge n \leq \mathit{max\text{-}nprocs\text{-}word} \Longrightarrow \mathit{nprocs\text{-}key}\ \mathit{OR}\ n \neq \mathit{nprocs\text{-}key}$ (**is** *?nbound*
$\Longrightarrow$ -)
**proof**−
**assume** *0*:*?nbound*
**hence** $\exists\ i{\in}\{0..{<}24\}.(\mathit{nprocs\text{-}key}\ !!\ i \vee n\ !!\ i) \neq \mathit{nprocs\text{-}key}\ !!\ i$
**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-key* doesn't belong to the set of procedure key addresses.

**lemma** *nprocs-key-notin-proc-key-addrs*: $\vdash \mathcal{S} \Longrightarrow \mathit{nprocs\text{-}key} \notin \mathit{proc\text{-}key\text{-}addrs}\ \mathcal{S}$
**using** *proc-id-bounded proc-key-addr-neq-nprocs-key*
**unfolding** *proc-key-addrs-def*
**by** *auto*

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

**definition** *proc-key-of-id* $\mathcal{S} \equiv$ *the-inv-into* (*proc-keys* $\mathcal{S}$) (*proc-id* $\mathcal{S}$)

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

**lemma** *proc-key-of-id-inv*[*simp*]: $[\![\vdash\mathcal{S};\ k \in \mathit{proc\text{-}keys}\ \mathcal{S}]\!] \Longrightarrow \mathit{proc\text{-}key\text{-}of\text{-}id}\ \mathcal{S}\ (\mathit{proc\text{-}id}$
$\mathcal{S}\ k) = k$
**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*: $[\![\vdash\mathcal{S};\ i\in\{1..nprocs\ \mathcal{S}\}]\!] \Longrightarrow \exists\, k\in proc\text{-}keys\ \mathcal{S}.\ proc\text{-}id\ \mathcal{S}\ k = i$
**proof** (*rule ccontr*, *subst* (*asm*) *bex-simps*(*8*))
  **let** $?rng = \{1\ ..\ nprocs\ \mathcal{S}\}$
  **let** $?prj = proc\text{-}id\ \mathcal{S}$ ‘ $proc\text{-}keys\ \mathcal{S}$
  **assume** $\forall\, k\in proc\text{-}keys\ \mathcal{S}.\ proc\text{-}id\ \mathcal{S}\ k \neq i$
  **hence** $0{:}i \notin\ ?prj$
    **by** *auto*
  **assume** $\vdash \mathcal{S}$
  **hence** $1{:}?prj \subseteq\ ?rng$ **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** $*{:}i\in ?rng$
  **have** $card\ ?prj = card\ (?prj \cup \{i\} - \{i\})$
    **using** *0* **by** *simp*
  **also have** ... $<$ $card\ (?prj \cup \{i\})$
    **by** (*rule card-Diff1-less*, *simp-all*)
  **also from** $*$ **have** ... $\leq$ $card\ ?prj$
    **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*: $[\![\vdash\mathcal{S};\ i\in\{1..nprocs\ \mathcal{S}\}]\!] \Longrightarrow proc\text{-}key\text{-}of\text{-}id\ \mathcal{S}\ i \in proc\text{-}keys\ \mathcal{S}$
  **using** *proc-key-exists the-inv-into-into*[*of proc-id* $\mathcal{S}$ *proc-keys* $\mathcal{S}$ *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**
  $con\text{-}wit\text{-}map\ \mathcal{S} :: \text{-} \rightharpoonup word32 \equiv$
      $[nprocs\text{-}key \mapsto of\text{-}nat\ (nprocs\ \mathcal{S})]$
    $++\ (Some \circ ucast \circ proc\text{-}key\text{-}of\text{-}id\ \mathcal{S} \circ id\text{-}of\text{-}key\text{-}addr)\ |‘\ proc\text{-}key\text{-}addrs\ \mathcal{S}$

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

**definition** $con\text{-}wit\ \mathcal{S} \equiv override\text{-}on\ zero\text{-}con\ (the \circ con\text{-}wit\text{-}map\ \mathcal{S})\ (dom\ (con\text{-}wit\text{-}map\ \mathcal{S}))$

**lemmas** *con-wit* = *con-wit-def con-wit-map-def comp-def*

**lemma** *restrict-subst*[*simp*]: $k \in S \Longrightarrow (f\ |‘\ \{\ g\ k \mid k.\ k \in S\})\ (g\ k) = f\ (g\ k)$
  **unfolding** *restrict-map-def*

**by** *auto*

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

**theorem** *models-nonvac*: $\vdash \mathcal{S} \Longrightarrow \exists\ S.\ S \Vdash \mathcal{S}$
**unfolding** *models-nprocs models-proc-keys*
**proof** (*intro exI*[*of - con-wit* $\mathcal{S}$] *conjI*)
**assume** *wf*:$\vdash\mathcal{S}$
**thus** *unat* (*con-wit* $\mathcal{S}$ *nprocs-key*) = *nprocs* $\mathcal{S}$
**unfolding** *con-wit*
**using** *nprocs-key-notin-proc-key-addrs le-unat-uoi*[**where** *z*=*max-nprocs-word*]
**by** (*subst override-on-apply-in*, *simp*)
(*subst map-add-dom-app-simps*(*3*), *auto simp add*:*procs-rng-wf*)
**from** *wf* **have** $\bigwedge k.\ k \in$ *proc-keys* $\mathcal{S} \Longrightarrow$
*proc-key-of-id* $\mathcal{S}$ (*id-of-key-addr* (*key-addr-of-id* (*proc-id* $\mathcal{S}$ *k*)))
= *k*
**by** (*subst id-of-key-addr-inv*)
(*auto simp add*:*procs-rng-wf*, *force*)
**thus** $\forall$ *k*$\in$*proc-keys* $\mathcal{S}$. *con-wit* $\mathcal{S}$ (*key-addr-of-id* (*proc-id* $\mathcal{S}$ *k*)) = *ucast k*
**unfolding** *con-wit proc-key-addrs-def*
**by** (*intro ballI*, *subst override-on-apply-in*, (*auto*)[*1*])
(*subst map-add-find-right*, *subst restrict-subst*, *auto*)
**qed**

# 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-key*) $\leq$ *max-nprocs-nat*

Well-formedness of procedure keys:

1. procedure keys should fit into 24-byte words;
2. they should represent some existing procedures (currently this essentially a temorary 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.

**definition** *proc-keys-wf* (*dummy* :: *'a itself*) (*S* :: *storage*) $\equiv$
($\forall$ *k*$\in${*S* (*key-addr-of-id i*) | *i*. *i*$\in${*1*..*unat* (*S nprocs-key*)}}. *ucast* (*ucast k* :: *key*) = *k*)

$\wedge$ ($\forall$ *i*$\in${*1..unat* (*S nprocs-key*)}. $\exists$ *p* :: (*'a* :: *proc-class*). *ucast* (*key p*) = *S* (*key-addr-of-id i*))
$\wedge$ *inj-on* (*S* $\circ$ *key-addr-of-id*) {*1..unat* (*S nprocs-key*)}

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

**definition** *con-wf* ($\Vdash$_- [*60*]) **where**
$\Vdash_{(d\ ::\ ('a\ ::\ proc\text{-}class)\ itself)}$ $S \equiv$
*nprocs-wf S*
$\wedge$ *proc-keys-wf d S*

**notation** (*input*) *con-wf* ($\Vdash$_- [*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*:
**assumes** $i \in \{1..unat\ (S\ nprocs\text{-}key)\}$
**assumes** $\vdash \mathcal{S}$
**assumes** $S \Vdash \mathcal{S}$
**obtains** *k* **where** $k \in proc\text{-}keys\ \mathcal{S} \wedge i = proc\text{-}id\ \mathcal{S}\ k$
**using** *assms proc-key-exists*
**unfolding** *models-nprocs*
**by** *metis*

**lemmas** *key-upcast* =
*ucast-down-ucast-id*
[**where** *?'b=256* **and** *?'a=192*,
*simplified is-down-def target-size-def source-size-def*]
*down-ucast-inj*
[**where** *?'b=256* **and** *?'a=192*,
*simplified is-down-def target-size-def source-size-def*]

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

**theorem** *model-wf*:$[\![\vdash(\mathcal{S}\ ::\ ('p\ ::\ proc\text{-}class)\ abs);\ S \Vdash \mathcal{S}]\!] \Longrightarrow \Vdash_{TYPE\ ('p)} S$
**unfolding** *proc-keys-wf*
**proof** (*intro conjI ballI*)

**assume** *wf*:$\vdash \mathcal{S}$ **and** *models*:$S \Vdash \mathcal{S}$
**thus** *nprocs-wf S*
**unfolding** *procs-rng-wf models-nprocs nprocs-wf-def* **by** *simp*
**show** *inj-on* ($S \circ$ *key-addr-of-id*) {*1..unat* (*S nprocs-key*)}
**unfolding** *inj-on-def*
**proof** (*intro ballI impI*)
**fix** *x y*
**assume** *x*∈{*1..unat* (*S nprocs-key*)} **and** *y*∈{*1..unat* (*S nprocs-key*)}
**from** *wf models* **and** *this*
**show** ($S \circ$ *key-addr-of-id*) *x* = ($S \circ$ *key-addr-of-id*) *y* $\Longrightarrow$ *x* = *y*
**using** *key-upcast*
**by** (*elim elim-proc-id*[**where** $\mathcal{S}$=$\mathcal{S}$]) (*auto simp add*:*models-proc-keys*)
**qed**
**fix** *i*
**assume** *i*∈{*1..unat* (*S nprocs-key*)}
**thus** ∃ *p* :: *'p*. *ucast* (*key p*) = *S* (*key-addr-of-id i*)
**using** *wf models*
**by** (*intro exI*[*of - proc-bdy* $\mathcal{S}$ (*proc-key-of-id* $\mathcal{S}$ *i*)], *elim elim-proc-id*)
(*simp*+, *simp add*:*models-proc-keys procs-rng-wf*)
**next**
**fix** *k*
**assume** *wf*:$\vdash \mathcal{S}$ **and** *models*:$S \Vdash \mathcal{S}$
**assume** *k*∈{*S* (*key-addr-of-id i*) | *i*. *i*∈{*1..unat* (*S nprocs-key*)}}
**thus** *ucast* (*ucast k* :: *key*) = *k*
**proof** (*simp only*:*Setcompr-eq-image image-iff*, *elim bexE*)
**fix** *x*
**assume** *x* ∈ {*1..unat* (*S nprocs-key*)} **and** *k* = *S* (*key-addr-of-id x*)
**thus** *ucast* (*ucast k* :: *key*) = *k*
**using** *wf models key-upcast*
**by** (*elim elim-proc-id*[**where** $\mathcal{S}$=$\mathcal{S}$])
(*auto simp add*:*models-proc-keys*)
**qed**
**qed**

# 7 Decoding of storage

Auxiliary abbreviations

**abbreviation** *proc-pair S i* ≡ (*ucast* (*S* (*key-addr-of-id i*)) :: *key*, (*i*, *SOME p*. *True*))

**abbreviation** *proc-list S* ≡ [*proc-pair S i*. *i* ← [*1..<Suc* (*unat* (*S nprocs-key*))]]

The decoding function:

**definition** *abs* (⦃-⦄) **where** ⦃*S*⦄ = (| *procs* = *map-of* (*proc-list S*) |)

**lemma** *inj-on-fst*: *inj-on f A* $\Longrightarrow$ *inj-on* (λ *x*. (*f x*, *y x*)) *A*
**unfolding** *inj-on-def* **by** *simp*

# 8 System calls

This section will contain specifications of the system calls, but for now there are only some early experiments.

**abbreviation** *higher32* **where** $higher32\ k\ n \equiv n >> ((32 - k) * 8)$

**definition** *is-kernel-storage-key* :: *word32* ⇒ *bool*
  **where** $is\text{-}kernel\text{-}storage\text{-}key\ w \equiv higher32\ 4\ w = 0xffffffff$

## 8.1 Register Procedure

**definition** *n-of-procedures* :: *storage* ⇒ *word32*
  **where** $n\text{-}of\text{-}procedures\ s = s\ nprocs\text{-}key$

**definition** *add-proc* :: *key* ⇒ *storage* ⇒ *storage option*
  **where**
    *add-proc p s* ≡
      *if n-of-procedures s* < *max-nprocs-word*
      *then*
        *Some* (*s*
            (*nprocs-key* := *n-of-procedures s* + *1*,
             *nprocs-key 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-key OR n-of-procedures s* + *1* ≠ *nprocs-key*
    **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**

# 9 Tests

These are tests that we use to quickly check that the implemented functions and lemmas are correct, before conducting full-scale proofs.

**lemma** (*0xFF* :: *word32*) = (*255* :: *word32*) **by** *auto*
**value** *is-kernel-storage-key 0xffffffff00000000000000000000000000000000000000000000000000000000*
**value** (*0b1011* :: *word32*) >> *2* = *0x02*
**value** *0b1111* :: (*2 word*)
**value** *ucast* (*0b1010* :: *4 word*) = (*0b101010* :: *6 word*)
**value** *case* (*add-proc 5* (λ*x.0*)) *of* (*Some s*) ⇒ *s nprocs-key*
**value** *case* (*add-proc 5* (λ*x.0*)) *of* (*Some s*) ⇒ *s* (*nprocs-key OR 1*)
**value** *case* (*add-proc 5* (λ*x.0*)) *of* (*Some s*) ⇒ *s* (*nprocs-key OR 2*)
**end**