
Well-behaved Memory on Top of Virtual Memory

Hendrik Tews

SOS Group, Radboud Universiteit Nijmegen

<http://www.sos.cs.ru.nl/~tews>

(with support from Marcus Völp, Michael Hohmuth, Shane G. Stephens)

Virtual memory is not well-behaved (inside an Operating system)

The following cannot be proved in general:

$$\{p \neq q\} *p = 0; *q = 1 \{ *p == 0 \}$$

because

- even if p and q point to different *virtual* addresses
- these virtual addresses might denote the same physical memory (or overlapping regions in physical memory)

Implications

- OS verification must include address translation and must go down to physical memory (at least for some parts of the OS)
- (non implication) application programs rely on OS to provide sane virtual memory

- I. Today's problem
- II. Project background
- III. General approach
- IV. Related problems
- V. The plain memory specification
- VI. Conclusion

Robin – verified trustworthy computing base for OS virtualization

- European project with TU Dresden, ST Microelectronics, Nijmegen University, Secunet
- Goal: to built a small robust trusted computing base
- permit to run several instances of legacy OS'es (in parallel with special applications)
- enforce complete separation (if wished)
(e.g, of a secure Email-client and the latest bells-and-whistle-web browser, which opens the door wide for any intruder)
- verification of selected properties of the kernel
- verification workpackage is more or less a continuation of VFiasco
- **See <http://robin.tudos.org/>**

I. Today's problem

II. Project background

III. General approach

IV. Related problems

V. The plain memory specification

VI. Conclusion

Split kernel verification?

- code of the kernel probably splits into 5% dirty and 95% well-behaved
- could use a simple semantics for the 95% well-behaved code and establish its assumptions by verifying the dirty 5%
- **Our approach:** Use *one* faithful and correct semantics for the complete kernel, which simplifies considerably in the well-behaved case

Technology

- We favour source code verification
- instead of model checking/type checking/...
- because this technique can catch all errors
- can be extended to object-code verification later

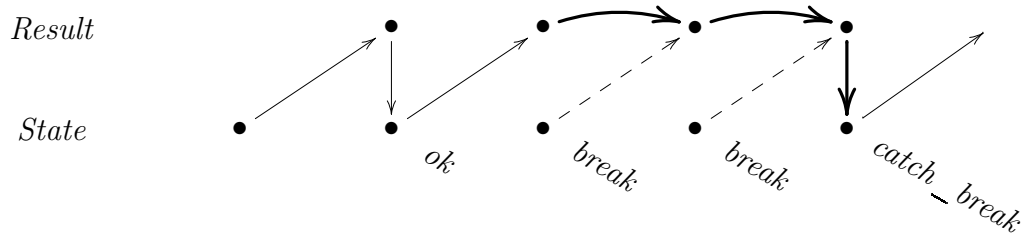
- semantic domain: State transformers

$$State \longrightarrow Result$$

- *State* is the set of all system states (describes memory contents, address translation, ...)
- *Result* is a disjoint union

$$Result = \begin{cases} State \uplus ok \text{ (normal termination)} \\ State \uplus break \\ \mathbf{1} \uplus \dots \text{ hang diversion} \end{cases}$$

- Intuition



- denotational semantics is based on a (abstract) hardware model (IA32) (symbolic evaluation of C++ inside PVS)

- I. Today's problem
- II. Project background
- III. General approach

- IV. Related problems**

- V. The plain memory specification
- VI. Conclusion

Read access can change the contents of the memory

- because of the page fault handler
- solution: model as state transformer, which is free to change the state

page fault handling is inherently recursive

- page fault handler is called even for page faults in the page fault handler
- **Our approach:** recursiveness of the page fault handler is bounded and very small (2 or 3)
 1. verify code that does never require a page fault handler (hopefully some parts of the page-fault handler)
 2. verify code that only needs a non-recursive page-fault handler (hopefully the complete real page-fault handler)
 3. verify the remainder

Verification will involve different memory models

- startup code (before address translation is switched on)
- because of the simple solution to recursive page-fault handlers
 - Memory access is actually a while loop with the page-fault handler as body
 - a change in the page-fault handler changes the memory model

- I. Today's problem
- II. Project background
- III. General approach
- IV. Related problems
- V. The plain memory specification**
- VI. Conclusion

Plain memory behaves just as you would expect

- plain memory describes a subset of the available (virtual) memory
- there is no virtual aliasing in this subset
- the property is preserved by certain, safe operations (plain memory forms an invariant)
- (the challenge lays in describing all this without referring to the address translation, because the plain memory abstraction shall be used with all kinds of memory models)

Minor generalisations

- split plain memory in read-only and read-write to permit reading the page directory without leaving the plain memory
- permit harmless additional operations, like
 - the semantics of `break`, `continue`, `goto`
 - switching the page directory while keeping the address translation of all plain memory constant (shared kernel page tables)

Leave and reestablish plain memory when

- the address translation of the plain memory is changed
- side effects in the page directory (dirty, accessed bits)

In detail: Plain memory consists of two sets of addresses, ro_addr and rw_addr , such that

1. reading in $ro_addr \cup rw_addr$ is always successful
2. reading in $ro_addr \cup rw_addr$ does not change the contents of $ro_addr \cup rw_addr$
3. writing in rw_addr is always successful
4. writing in rw_addr does not change ro_addr
5. writing in rw_addr does change the target of the write operation in rw_addr in the expected way and leaves the remainder unchanged
6. reads in ro_addr and read/writes in rw_addr preserve the plain-memory property
7. allow for a set of other actions (i.e., state transformers) that
 - do not change $ro_addr \cup rw_addr$
 - do preserve the all these properties

Technically two sets ro_addr and rw_addr form a plain memory with respect to

- a set of states
- a memory model, given as a read and a write operation
- a set of “safe” operations
(derived from ro_addr and rw_addr and explicitly provided in point 7)

```
pm : Var Memory_struct[State]
states : Var PRED[State]
ro_addr, rw_addr : Var PRED[Address]
other_actions : Var PRED[[State -> Result[State, Unit]]]

plain_memory?(pm, states, ro_addr, rw_addr, other_actions) : bool =
  unchanged_memory_invariant?(pm, states,
    union(other_actions, memory_read_transformers(pm,
      union(ro_addr, rw_addr))), union(ro_addr, rw_addr))
AND
  unchanged_memory_invariant?(pm, states,
    memory_write_transformers(pm, rw_addr),
    ro_addr)
AND
  unchanged_memory_write_invariant?(pm, states, rw_addr)
AND
  changed_memory_invariant?(pm, states, rw_addr)
AND
  transformers_ok?(states,
    union(memory_read_transformers(pm, union(ro_addr, rw_addr)),
      memory_write_transformers(pm, rw_addr)))
```

```
pm : Var Memory_struct[State]
states : Var PRED[State]
ro_addr, rw_addr : Var PRED[Address]
other_actions : Var PRED[[State -> Result[State, Unit]]]

dt : Var (interpreted_data_type?[Data])
s : Var State
addr : Var Address
data : Var Data

ad_write_ok : Lemma
  plain_memory?(pm, states, ro_addr, rw_addr, other_actions) And
  states(s) And
  in_plain_memory?(dt, addr, rw_addr)
  Implies
    OK?((
      write_data(pm,dt)(addr, data) ##
      read_data(pm,dt)(addr)
    )(s) )
```

```
read_write_res : Lemma
  plain_memory?(pm, states, ro_addr, rw_addr, other_actions) And
  states(s) And
  in_plain_memory?(dt, addr, rw_addr)
  Implies
    get_data((
      write_data(pm,dt)(addr, data) ##
      read_data(pm,dt)(addr)
    )(s))
    = data
```

Assuming we have

- a plain memory
- a starting state s , belonging to the plain memory
- a C++ data type, given by dt
- an address $addr$, such that dt fits into the plain memory at $addr$

then, it holds that

- reading something of type dt at $addr$
- directly after writing data at $addr$
- will terminate successfully (lemma on previous slide)
- deliver the data written

read_write_other_res : **Lemma**

plain_memory?(pm, states, ro_addr, rw_addr, other_actions) **And**

states(s) **And**

in_plain_memory?(dt1, addr1, rw_addr) **And**

in_plain_memory?(dt2, addr2, union(ro_addr, rw_addr)) **And**

blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) **And**

valid_in_mem(pm, dt2)(addr2)(s)

Implies

get_data((
 write_data(pm, dt1)(addr1, data1) ##
 read_data(pm, dt2)(addr2)
))(s)

=

get_data(read_data(pm, dt2)(addr2)(s))

If the memory blocks are disjoint, writing at addr1 does not affect the data at addr2.

As expected

```
phy_mem_plain_memory : Lemma  
  plain_memory?(phy_pm, fullset[Physical_memory], emptyset[Address],  
    physical_addresses(max), emptyset[[Physical_memory→Result[Physical_memory, Unit]]])
```

Physical memory forms a plain memory with all physical addresses being read-write enabled.

Still to do

Virtual memory forms a plain memory if

- there is no virtual aliasing among the addresses
- the page directory/page tables are not contained in the *rw_addr*
- the page-fault handler behaves reasonable (does not touch plain memory)

- I. Today's problem
- II. Project background
- III. General approach
- IV. Related problems
- V. The plain memory specification

- VI. Conclusion**

- plain memory abstraction captures well-behaved memory
- theorems about plain memory serve as a short cut in the fairly complicated model of virtual memory
- hopefully, the 95% clean OS code can be verified with just using the plain memory theorems