A Simple Virtual Memory Scheme Formalized in IMPS*

Joshua D. Guttman

The MITRE Corporation

Abstract. In this paper we formalize a simple virtual memory scheme derived from Mach and Multics. It is represented by state machine operations in IMPS, an Interactive Mathematical Proof System. We prove that a store with a global page table faithfully refines an abstract machine in which processes may perform fetch and store operations against a set of persistent memory objects. Both safety conditions and liveness conditions are proved. Some fine points treated in the model include: the finiteness of virtual address spaces and physical store; the ability to map a portion of a permanent memory object into the address space; initialization of newly allocated memory to zero; and the need for page-aligned addresses in some operations.
The paper has two main purposes. First, it illustrates how naturally IMPS can model this sort of problem, can prove the necessary theorems, and can present the results. Technical details are presented as typeset automatically by IMPS. Second, it illustrates how virtual memory systems can be specified and, at least at the first refinement levels, verified. Lower refinement levels would, however, raise additional issues of concurrency and of hardware dependencies.

1 Introduction

IMPS, an Interactive Mathematical Proof System [6], aims to provide mechanical support for traditional methods and activities of mathematics, and for traditional styles of classical mathematical proof. The bulk of IMPS work has focused on mathematics [7, 5]. However, the same broadly understandable techniques are also valuable for formal methods. This paper will illustrate that IMPS provides an attractive and flexible modeling framework for formal methods, and that IMPS provides adequate interactive theorem proving power to expose errors in specifications and to prove correct refinements.

A second goal is to describe a style in which virtual memory systems can be specified, and, at least at the highest refinement levels, verified. Our motivation for doing so derives from a recently begun effort (called VMACH) which is devoted

* Supported by the United States Army CECOM under contract DAAB07-94-C-H601. The development of IMPS was supported by the MITRE-Sponsored Research program. Author's address: The MITRE Corporation, 202 Burlington Rd, Bedford MA 01730-1420 USA; E-mail: `guttman@mitre.org`; Telephone: 617-271-2654; Fax: 617-271-3816.

to specifying and verifying critical subsytems of the Mach microkernel [10]; initial emphasis is on the Mach virtual memory system. Our reasons for undertaking this are partly research goals, and partly driven by the applications needs.

The research goal for VMACH is to study techniques for applying formal methods to portions of complex, multi-threaded software. Mach offers clever optimizations, such as the copy-on-write optimization, that interrelate virtual memory and inter-process communication [10, Chapters 4–5]. It also allows user-level processes called external memory managers to participate in virtual memory management, and to offer persistent memory objects that processes can map directly into their address spaces. The protocols between the Mach kernel and the external memory managers raise concurrency control problems [11]. But there is also a more immediately practical motivation for VMACH.

Mach is considered a promising architecture for workstations and other end systems in networks for which assurance is critical. These may include, for instance, networks processing financial transactions, or multiple levels of classified information, or air-traffic control information. Mach's microkernel architecture is advantageous because it allows many different applications-specific facilities, including widely varying security policies, to be built on top of a single kernel. The basic services of this kernel may be designed, verified and implemented once. The cost of applying formal methods may thus be amortized over a wide range of high-assurance applications.

The modest-sized example presented in this paper, which one person developed, debugged, and completely proved during a three week period, touches on only a few of the issues VMACH will eventually face.[2] In particular, this paper will not consider concurrency problems. Rather, it describes a sequential specification, to which a well-behaved multi-threaded implementation may conform. Moreover, the example takes a form more reminiscent of Multics's style of virtual addressing [12] than of Mach's.

IMPS as a Specification language. IMPS supports mathematics, and formal methods, using the "little theories" version of the axiomatic method [5]. The IMPS user develops a collection of axiomatic theories, all within a single fixed logic. Theories may be related in two main ways: a theory may *extend* a number of other theories, and a *theory interpretation* may translate one theory into another. A particular case study may introduce several theories; smaller, more modular theories encourage re-use and allow a gradual development from an abstract presentation to a more detailed specification. The example in this paper introduces four new theories. By contrast, a more detailed IMPS formalization of the Mach system state (based on [1]) introduces a dozen in 15 pages of text.

The fixed IMPS logic is intended not to be novel, but rather to provide a natural framework for formalizing classical mathematics. We have selected a version of classical simple type theory, which offers a convenient notation for functions and a syntactic "type-checking" discipline in its type system.

[2] For information on how VMACH relates to other formal methods work at The MITRE Corporation, see [8].

We have, however, introduced two characteristics which are more unusual in classical logics [3, 4]. These two areas represent ways that the logical tradition appears to have diverged from the dominant style of careful rigorous mathematics.

First, in IMPS the functions occurring in higher types include *partial* functions rather than just total functions. Partial functions are treated in a direct way: if, for instance, the value that t denotes is not in the domain of the function that f denotes, then the expression $f(t)$ simply has no denotation.

Second, within the framework of simple type theory we have introduced a mechanism of *sub-types*. For instance, if in some theory $\mathbf{R}$ is a type representing the real numbers, $\mathbf{Q}$ and $\mathbf{Z}$ may be sub-types representing the rationals and the integers. Any non-empty subset of a type may be distinguished as a sub-type. Types and their sub-types are jointly called *sorts*.

These two characteristics fit together quite smoothly in simple type theory. A function type, $\mathbf{Z} \rightharpoonup \mathbf{Q}$, for instance, represents the set of partial functions with domain included in $\mathbf{Z}$ and range included in $\mathbf{Q}$. It is a subtype of $\mathbf{R} \rightharpoonup \mathbf{R}$.

The IMPS logic is nevertheless classical in its treatment of formulas; for instance, a formula is a (propositional) tautology in the IMPS logic if and only if it is a classical tautology. We arrange this by two main conventions:

1. The denotation of any *predicate* is a total function into $\{$*True*, *False*$\}$;
2. An *atomic* formula is false if any of its immediate constituents has no denotation.

Intuitive Content of the Specification. The specification describes the service which a virtual memory system provides to the user-level processes executing on the system. This service makes available a set of potentially persistent *memory objects* which one or more processes can include directly within their virtual address space. A process is said to *map* a memory object into its address space when it allocates a range of virtual addresses to be used to reference the contents of that memory object. When a process maps a memory object, it specifies what part of its virtual address space it wishes to use for this purpose.

We will follow Multics terminology and refer to the part of the address space allocated in a single map operation as a *segment*.[3] The segments in an address space may be identified by number, and a simple addressing scheme, derived from Multics, is to regard the upper bits of a virtual address as determining a segment number s, while the lower bits determine an offset o within that portion of memory. This addressing scheme may be regarded as a two-dimensional arrangement, in which a word is accessed by a segment/offset pair $\langle s, o\rangle$.

A map operation takes as arguments not only the process p performing it, the desired segment number s, and the memory object m to be mapped, but also two additional numbers. These are a *base* b, which is the index within the

[3] In Multics a segment had a (quite small) maximum size, and the whole issue was intricately intertwined with hardware considerations [12]. However, neither of these facts is essential, and neither is relevant to Mach. See, e.g., [13, Section 3.7] for a recent general discussion of segmented memory systems.

memory object at which the mapped portion will begin; and a *length* ℓ, which is the number of words to map. The process will have access to the ℓ words beginning with the bth word contained in the object m. The virtual address $\langle s, o\rangle$ will reference the word $b + o$ within m if $o < \ell$; otherwise the reference is illegal and will cause a segmentation error.

In addition to the *map* operation, our specification will also offer *unmap*, *fetch*, and *store*. The *unmap* operation deallocates a segment s, so that addresses $\langle s, o\rangle$ will be henceforth illegal. A *fetch* communicates the value of the word associated with the virtual address $\langle s, o\rangle$ to the user process p, while *store* allows p to communicate a new value w which will henceforth occupy the $(b + o)$th position within m.

Thus our example provides only four typical operations, rather than the dozen that would be needed for a full description of the virtual memory interface to Mach; the state is similarly simplified. The Multics-style two-dimensional addressing also makes it easier to determine what memory object has been referenced by a virtual address.

The state itself has four components, each of them a partial function. The first three components, α, β, and Λ, take as arguments a process p and a segment s. They return, respectively, the memory object (if any) mapped by p in segment s, the base of the mapped portion, and the length of the mapped portion.[4] The operations respect an invariant regarding these three state components: if σ is an accessible state, then if any one of the three terms $\alpha(\sigma)(p, s)$, $\beta(\sigma)(p, s)$, and $\Lambda(\sigma)(p, s)$ is well-defined, then all three of them are well-defined. When these terms are not well-defined, that means that p has no object mapped in the particular segment s. Thus the domain of the partial functions expresses important state information.

The fourth state component γ is a partial function taking as arguments a memory object m and a natural number index i. The value $\gamma(\sigma)(m, i)$, if well-defined, returns the word at the ith location in m, in state σ. Nothing in the specification entails that the unary function $\lambda i \,.\, \gamma(\sigma)(m, i)$ is defined on an initial segment of the natural numbers. For instance, to represent a common architecture, of which MIPS is an example, in which bytes are addressable but fetches are made only from word-aligned addresses, we could specify it to be well-defined only if $i \bmod 4 = 0$.

Modeling Virtual Memory. The main ingredient in a virtual memory scheme is that the operating system uses the primary physical store ("main memory") as a cache for parts of the address space of processes. When a process references a location that is represented in the store, the operating system translates the virtual address to determine the page of store and the offset within that page to access. Hardware and operating system software cooperate in this translation. When a process attempts to reference a location that is not represented in the store, it takes a page fault. The operating system attempts to page the needed

[4] We could alternatively have had a single state component which returned a triple when defined, containing memory object, base, and length.

portion of the address space in from secondary store. To do so, it may need to free a page in primary store by flushing its current contents out to secondary store. When these maneuvers are complete, the operating system restarts the process at the same instruction that caused the page fault.

To justify using virtual memory to provide the services we have just described, we must extend the notion of state. In our approach, we will enrich it with a single global page table π, which determines what page (if any) in primary store represents a portion of a persistent memory object. We will also add a physical store μ, which, given a page and an offset within it, determines the word physically present at that location. Thus an implementation state σ_i has three components, namely *vsig*, π, and μ, which represent the embedded virtual memory specification state, the global page table, and the physical memory of the system respectively.

An implementation fetch or store then uses π to determine whether the requested virtual address is resident. The current store contents μ determines the word fetched or the function to be updated by the assignment, respectively. An implementation fetch or store cannot occur if the page is not resident. The implementation offers two additional operations, *page_in* and *page_out*, to cause a page's worth of data to become resident, and to clear a page's worth of store for new data.

The Abstraction Function. The states of the implementation are related to those of the specification using a traditional *abstraction function* [9]. For any implementation state, there is at most one specification state it represents, although the same specification state may be represented by many implementation states.

The heart of the abstraction function *abstr* is to overlay the physical store contents in $\mu(\sigma_i)$ over the "disk copies" of the memory objects, $\gamma(\mathrm{vsig}(\sigma_i))$. The auxiliary function γ_{abstr} uses the page table π to do so. The rest of the abstract state contains values from the embedded state $\mathrm{vsig}(\sigma_i)$.

When f, a function on implementation states, is a composition of state machine operations, we will call it an *abstract no-op* when $\mathrm{abstr}(f(\sigma_i)) = \mathrm{abstr}(\sigma_i)$.

The Main Theorems. We include a few theorems that indicate that the operations have been specified properly. However, the main theorems are refinement theorems. They establish a safety condition and a liveness condition. Suppose first that g is one of the four specified operations, with parameters fixed, and h is the implementation version of that operation. Then the safety condition states that

$$h(\sigma_i)\downarrow \Rightarrow \ \mathrm{abstr}(h(\sigma_i)) = g(\mathrm{abstr}(\sigma_i)),$$

where $\downarrow$ means "is defined."

To state the liveness constraints we need to define the assertion $s \simeq t$ (read "s is quasi-equivalent to t"). It abbreviates the assertion $(s\downarrow \vee t\downarrow) \Rightarrow s = t$. Quasi-equivalence says that the terms have the same denotation or lack thereof, and in the IMPS logic it is the condition that justifies substitution of s for t, wherever s is free for t.

The liveness condition states that there is an abstract no-op f such that

$$\text{abstr}(h(f(\sigma_i))) \simeq g(\text{abstr}(\sigma_i)).$$

We call this a liveness condition because it entails that when the abstract machine can undergo operation g from $\text{abstr}(\sigma_i)$, then even if the implementation cannot undergo h from σ_i, it can evolve to an indistinguishable state $f(\sigma_i)$ from which it can undergo h. In our case study, the abstract no-op f consists of a *page_in* operation, if needed, preceded by a *page_out* operation, if needed.

If by contrast h is *page_in* or *page_out* operation, to which nothing in the abstract version corresponds, then the safety condition is that h is an abstract no-op, $\text{abstr}(h(\sigma_i)) = \sigma_i$. There is no liveness constraint for this case.

These properties entail that every computation[5] of the abstract state machine corresponds to a computation of the implementation machine and *vice versa*, in the following sense of "correspond." If C_i is a computation of the implementation, then an abstract computation C *corresponds* to C_i if there is a non-decreasing function $f : \mathbf{N} \rightharpoonup \mathbf{N}$ onto the domain of C such that $\text{abstr}(C_i(j)) \simeq C(f(j))$.

Remainder of this Paper. The remainder of this paper consists of IMPS-generated typeset output together with explanatory prose. All of the **Theory**, **Definition**, **Theorem**, and similar environments have been constructed by IMPS directly from the fully verified source files. In a few places linefeeds or indentation have been added manually to the formulas. Many auxiliary lemmas have been deleted, partly to save space and partly to highlight the main theorems for the reader.

We emphasize the IMPS LaTeX facility because we think it important that a formal method provide a comprehensible transcript of the specifications and theorems that have been developed.

2 Virtual Memory Specification

2.1 Basic Vocabulary and State Definition

We begin by introducing a formal language or signature to use to talk about the virtual memory specification and its operations. This language extends the IMPS language for real arithmetic. It contains four additional basic types, namely *mem_obj*, *word*, *process*, and *page*. It also distinguishes two sub-sorts of the natural numbers, one to be used as segment numbers, the other to be used for offsets within segments. A virtual address is in effect a pair $\langle s, o\rangle$ for some segment number s and offset o. The two subsorts are unspecified, so that this specification is consistent with with a wide range of constraints on segment numbers and offsets. For instance, the sorts can be interpreted as finite, and the allowable values may be restricted to have an alignment property, such as being divisible by 2^n.

[5] That is, a finite or infinite sequence C of states such that $C(0)$ is an initial state, and for every j where $C(j+1)\downarrow$, there is an operation h such that $C(j+1) = h(C(j))$.

Two new individual constants are also introduced, namely *null_word* and *pagesize*, which are used respectively to represent the value to be inserted in newly initalized memory and the hardware-determined size of a page frame.

The "language" definition that follows—along with all "definitions," "sort definitions," "theorems," and similar environments—is generated automatically by IMPS. The parenthesized identifier is the name of the item being introduced, in this case, a new language.

Language 2.1 (vm-language)
Embedded language: *h-o-real-arithmetic*
Base types: mem_obj word process page
Sorts: seg $\ll$ **N** off $\ll$ **N**
Constants: null_word *:* word pagesize *:* off

We will need to assume that `pagesize` is not 0, so that page-aligned addresses will make sense in Section 3.

Component theory: h-o-real-arithmetic
Top level axioms:

pagesize-non-zero $\neg(\text{pagesize} = 0)$.

Fig. 1. Components and axioms for vm-1

Theory 2.2 (vm-1)
Language: *vm-language*
Component Theories and Axioms: *See Figure 1.*

We now introduce a freely generated datatype (or "BNF") into IMPS. As a consequence, IMPS constructs a new theory extending *vm-1*, which will be named *vm-spec*. We will work within this theory throughout the remainder of this section.

The new objects will belong to a logical type called *pre_vstate*. In general, a BNF may have a number of atoms, as well as a number of constructors, which may recursively construct new datatype elements from old. In this case, there are no atoms and only a single non-recursive constructor named *make_vstate*, so that the new datatype is effectively a tuple type. The four selectors extract the tuple components. IMPS constructs the expected "no junk" (induction) and "no confusion" axioms, and equips the theory with a good deal of other structure which is useful when the datatype definition is recursive.

Data Type Theory 2.3 (vm-spec)
Component Theory: *vm-1*
Primary Type: pre_vstate
Constructor:

make_vstate : [[process $\times$ seg $\rightharpoonup$ mem_obj] $\times$ [process $\times$ seg $\rightharpoonup$ $\mathbf{N}$] $\times$ [process $\times$ seg $\rightharpoonup$ off] $\times$ [mem_obj $\times$ $\mathbf{N}$ $\rightharpoonup$ word] $\rightharpoonup$ pre_vstate], *with selectors:*
- α : [pre_vstate $\rightharpoonup$ [process $\times$ seg $\rightharpoonup$ mem_obj]]
- β : [pre_vstate $\rightharpoonup$ [process $\times$ seg $\rightharpoonup$ $\mathbf{N}$]]
- Λ : [pre_vstate $\rightharpoonup$ [process $\times$ seg $\rightharpoonup$ off]]
- γ : [pre_vstate $\rightharpoonup$ [mem_obj $\times$ $\mathbf{N}$ $\rightharpoonup$ word]]

A *pre_vstate* is in effect the quadruple of its α, β, Λ, and γ components. Of these, $\alpha(\sigma)(p, s)$ is the memory object that p has mapped in segment s, $\beta(\sigma)(p, s)$ is the base at which its mapped portion begins, $\Lambda(\sigma)(p, s)$ is the length of the mapped portion, and $\gamma(\sigma)(m, i)$ returns the ith word contained in m.

Among the *pre_vstate*s, we are only concerned with ones where the α, β, and Λ components have the same domain. We segregate them into a sort *vstate*. The accessible states of our machine belong to this sort. Some work must be done to show that an operation such as *map* preserves this invariant, and returns a value within the subsort *vstate*. But this work must generally still be done using other approaches to formalization; in IMPS one then benefits from the system's ability to use sorting information effectively in reasoning about the domain and range of functions [6].

The sort definition that follows, generated automatically by IMPS, introduces a subsort of *pre_vstate* named *vstate*. The theory being extended by this definition is *vm-spec*. The members of the new sort are those satisfying the predicate that follows, namely those σ :*pre_vstate* such that the conjunction of the two bulleted assertions holds true. The notation [*args* $\mapsto$ *body*] is IMPS's presentation of the λ-expression λ*args* . *body*. This bracket notation can be nested; for instance

$$[\, x : \mathbf{R} \mapsto [\, y : \mathbf{R} \mapsto x + y \,]\,]$$

represents the curried function that, given x, returns the function that adds x to its argument. The word "conjunction" preceding bulleted items should be read "The following conjunction holds;" as the word "implication" should be read "The following implication holds;" and the biconditional sign $\iff$ should be read "The following are equivalent."

Sort Definition 2.4 (vstate) *Theory: vm-spec*
$[\,\sigma$: pre_vstate $\mapsto$
conjunction
- $\forall p$: process, s : seg $\quad \alpha(\sigma)(p, s) \downarrow \iff \beta(\sigma)(p, s) \downarrow$
- $\forall p$: process, s : seg $\quad \alpha(\sigma)(p, s) \downarrow \iff \Lambda(\sigma)(p, s) \downarrow]$.

2.2 Some Auxiliary Definitions

The following function resolves a p, s, o triple in the current vstate to an index into the memory object. In this definition, the parenthesized identifier *res_off* is defined to be equal to the expression which follows. In this case and most others, that expression is a function. Thus *res_off* is introduced as a function constant of sort [vstate $\times$ process $\times$ seg $\times$ off $\rightharpoonup$ $\mathbf{N}$]. In this definition, the $\perp\mathbf{N}$ is a term with no denotation, but which has syntactic sort $\mathbf{N}$. It ensures that *res_off* is defined only if $o < \Lambda(\sigma)(p,s)$.

Definition 2.5 (res_off) *Theory: vm-spec*
$[\,\sigma : \text{vstate}, p : \text{process}, s : \text{seg}, o : \text{off} \;\mapsto$
  conditionally, if $o < \Lambda(\sigma)(p,s)$
- *then* $\beta(\sigma)(p,s) + o$
- *else* $\perp\mathbf{N}\,]$.

The predicate *ref_legal* characterizes when an s, o pair represents a legal virtual address for the process p. It requires that p should have something mapped in segment s, and also that the offset o is not too large.

Definition 2.6 (ref_legal) *Theory: vm-spec*
$[\,\sigma : \text{vstate}, p : \text{process}, s : \text{seg}, o : \text{off} \;\mapsto$
  conjunction
- $\alpha(\sigma)(p,s)\downarrow$
- $o < \Lambda(\sigma)(p,s)\,]$.

The function *assign*$_v$ is defined to return an altered version of the function given as its last argument c; the result differs in that its value for $\langle m, i\rangle$ is w. It is used to define the *store* operation.

Definition 2.7 (assign$_\mathbf{v}$) *Theory: vm-spec*
$[\,m : \text{mem\_obj}, i : \mathbf{N}, w : \text{word}, c : \text{mem\_obj} \times \mathbf{N} \rightharpoonup \text{word} \;\mapsto$
  $[\,m_1 : \text{mem\_obj}, j : \mathbf{N} \;\mapsto$
    if $m_1 = m \wedge i = j$ *then* w *else* $c(m_1, j)\,]\,]$.

2.3 The Four Operations of the Specification

The Store Operation The definition of the *store* operation applies *assign*$_v$ to obtain the contents component γ for the resulting state. The functions α and res_off are used to return the memory object and index to alter.

Definition 2.8 (store) *Theory: vm-spec*
$[\,\sigma : \text{vstate}, p : \text{process}, s : \text{seg}, o : \text{off}, v : \text{word} \;\mapsto$

$$\text{make\_vstate} : \begin{cases} \alpha(\sigma), \\ \beta(\sigma), \\ \Lambda(\sigma), \\ \text{assign}_v(\alpha(\sigma)(p,s), \text{res\_off}(\sigma,p,s,o), v, \gamma(\sigma)) \end{cases}$$

$]$.

The *ref* auxiliary function determines a user process's view of the contents of memory. It uses α to determine the object referenced, and then uses *res_off* to retrieve an index into the object. If the object is undefined for the index, *ref* specifies that the user process sees null-filled initialized memory. This is the behavior of Mach's *vm_allocate* kernel operation [10].

Definition 2.9 (ref) *Theory: vm-spec*
$[\sigma : \text{vstate}, p : \text{process}, s : \text{seg}, o : \text{off} \mapsto$
conditionally
- *if* $\gamma(\sigma)(\alpha(\sigma)(p, s), \text{res\_off}(\sigma, p, s, o)) \downarrow$ *then* $\gamma(\sigma)(\alpha(\sigma)(p, s), \text{res\_off}(\sigma, p, s, o))$
- *else if* $\text{ref\_legal}(\sigma, p, s, o)$ *then* null_word
- *otherwise* $\perp\text{word}]$.

The *ref* function allows us to state a pair of correctness theorems for *store*, of which we will show only the first here. It states that when we reference the same location which has previously been stored, we retrieve the stored value. The omitted theorem states that when we reference a different location, we retrieve the same value that the previous state σ gave.

Theorem 1. (store-ref-same) *Theory: vm-spec*
$\forall\sigma : \text{vstate}, p_0, p_1 : \text{process}, s_0, s_1 : \text{seg}, o_0, o_1 : \text{off}, v : \text{word}$ implication
- conjunction
  - $\text{ref\_legal}(\sigma, p_0, s_0, o_0)$
  - $\text{ref\_legal}(\sigma, p_1, s_1, o_1)$
  - $\alpha(\sigma)(p_1, s_1) = \alpha(\sigma)(p_0, s_0)$
  - $\text{res\_off}(\sigma, p_1, s_1, o_1) = \text{res\_off}(\sigma, p_0, s_0, o_0)$
- $\text{ref}(\text{store}(\sigma, p_0, s_0, o_0, v), p_1, s_1, o_1) = v$.

The Fetch Operation We have arranged that all of the four operations are formalized as functions yielding the resulting state as their values. In the case of *fetch*, there is of course also another relevant piece of information, namely the value communicated from virtual memory to the process executing the *fetch*. We treat this as a *parameter* to the operation. The fetch returns the current state unchanged if this parameter has the right value, and is undefined if it is wrong. Thus, the operation is like a predicate which determines whether the value is correct or incorrect.

This approach was suggested by CSP's non-directional view of communication events (see [2, Section 4.1] for a similar example). In fact, the state machine presented here can be easily and naturally transformed into a CSP-style process description. This is desirable because it sketches a specification for an ultimately multi-threaded implementation.

Definition 2.10 (fetch) *Theory: vm-spec*
$[\sigma : \text{vstate}, p : \text{process}, s : \text{seg}, o : \text{off}, v : \text{word} \mapsto$
conditionally, if $v = \text{ref}(\sigma, p, s, o)$
- *then* σ
- *else* $\perp\text{vstate}]$.

The Unmap and Map Operations We use an auxiliary function to cause state components to become undefined for $\langle p, s\rangle$.

Definition 2.11 (free_seg) *Theory: vm-spec*
$[p : \text{process}, s : \text{seg} \mapsto$
$\quad [p_1 : \text{process}, s_1 : \text{seg} \mapsto$
$\quad\quad \mathit{if}\ p_1 = p \wedge s_1 = s\ \mathit{then}\ \bot\text{seg}\ \mathit{else}\ s_1\,]\,].$

Unmap is then straightforward to define, and *map* is equally predictable.

Definition 2.12 (unmap) *Theory: vm-spec*
$[\sigma : \text{vstate}, p : \text{process}, s : \text{seg} \mapsto$

$$\text{make\_vstate} : \begin{cases} [p_1 : \text{process}, s_1 : \text{seg} \mapsto \alpha(\sigma)(p_1, \text{free\_seg}(p,s)(p_1,s_1))\,], \\ [p_1 : \text{process}, s_1 : \text{seg} \mapsto \beta(\sigma)(p_1, \text{free\_seg}(p,s)(p_1,s_1))\,], \\ [p_1 : \text{process}, s_1 : \text{seg} \mapsto \Lambda(\sigma)(p_1, \text{free\_seg}(p,s)(p_1,s_1))\,], \\ \gamma(\sigma) \end{cases}$$

$].$

Definition 2.13 (map) *Theory: vm-spec*
$[\sigma : \text{vstate}, p : \text{process}, s : \text{seg}, b : \mathbf{N}, \text{len} : \text{off}, m : \text{mem\_obj} \mapsto$

$$\text{make\_vstate} : \begin{cases} [p_1 : \text{process}, s_1 : \text{seg} \mapsto \mathit{if}\ p_1 = p \wedge s_1 = s\ \mathit{then}\ m\ \mathit{else}\ \alpha(\sigma)(p_1,s_1)\,], \\ [p_1 : \text{process}, s_1 : \text{seg} \mapsto \mathit{if}\ p_1 = p \wedge s_1 = s\ \mathit{then}\ b\ \mathit{else}\ \beta(\sigma)(p_1,s_1)\,], \\ [p_1 : \text{process}, s_1 : \text{seg} \mapsto \mathit{if}\ p_1 = p \wedge s_1 = s\ \mathit{then}\ \text{len}\ \mathit{else}\ \Lambda(\sigma)(p_1,s_1)\,], \\ \gamma(\sigma) \end{cases}$$

$].$

3 A Rudimentary Implementation

3.1 Preliminaries: Page Offsets and Page-aligned Addresses

The function *align* returns the largest page-aligned address less than its argument i, while ρ returns the difference between i and the page-aligned value. A variety of arithmetical lemmas about these were proved. The sort of page aligned natural numbers is introduced as the set of natural numbers i such that $\text{align}(i) = i$. 0 was provided to IMPS as witness that this set is non-empty.

Definition 3.1 (rho) *Theory: vm-spec*
$[i : \mathbf{N} \mapsto$
$\quad i \bmod \text{pagesize}\,].$

Definition 3.2 (align) *Theory: vm-spec*
$[i : \mathbf{N} \mapsto$
$\quad \text{div}(i, \text{pagesize}) \cdot \text{pagesize}\,].$

Sort Definition 3.3 (aligned) *Theory: vm-spec*
$[i : \mathbf{N} \mapsto$
$\quad \text{align}(i) = i\,].$

3.2 The State of the Implementation

The following BNF form introduces another tuple type. Here, one component of the tuple is an embedded *vstate*. The other two components represent a global page table π (defined only for page-aligned indices) and a store μ.

Data Type Theory 3.4 (vm-impl)
Component Theory: *vm-spec*
Primary Type: pre_istate
Constructor:

make_istate : [vstate $\times$ [mem_obj $\times$ aligned $\rightharpoonup$ page] $\times$ [page $\times$ $\mathbf{N}$ $\rightharpoonup$ word] $\rightharpoonup$ pre_istate], *with selectors:*
vsig : [pre_istate $\rightharpoonup$ vstate]
π : [pre_istate $\rightharpoonup$ [mem_obj $\times$ aligned $\rightharpoonup$ page]]
μ : [pre_istate $\rightharpoonup$ [page $\times$ $\mathbf{N}$ $\rightharpoonup$ word]]

The implementation state obeys a crucial invariant, namely that $\pi(\sigma_i)$ has an injective function as its value. If this were not the case, then when we altered a single location in physical store, we would in effect have changed more than one abstract memory object location. Hence, in that case the implementation store operation would be unfaithful. We canonize this invariant in the definition of the sub-sort *istate*.

Sort Definition 3.5 (istate) *Theory: vm-impl*
$[\sigma_i : \text{pre\_istate} \mapsto$
$\forall m_0, m_1 : \text{mem\_obj}, a_0, a_1 : \text{aligned} \quad s.\ t. \quad \pi(\sigma_i)(m_0, a_0) = \pi(\sigma_i)(m_1, a_1),$
conjunction
- $m_0 = m_1$
- $a_0 = a_1\,].$

The Abstraction Function. Suppose m is a memory object and i is an index into it. We regard $\langle m, i\rangle$ as resident in physical store in state σ_i when $\langle m, \text{align}(i)\rangle$ is in the domain of the function $\pi(\sigma_i)$.

Definition 3.6 (resident) *Theory: vm-impl*
$[\sigma_i : \text{istate}, m : \text{mem\_obj}, i : \mathbf{N} \mapsto$
$\pi(\sigma_i)(m, \text{align}(i)) \downarrow].$

To determine the abstract contents of memory objects determined by an implementation state σ_i, we combine physical store with the disk copies. If an address is resident, then π and μ determine its current value. Otherwise, the contents function γ of the embedded *vstate* determines its value.

Definition 3.7 (gamma_abstr) *Theory: vm-impl*
$[\sigma_i : \text{istate} \mapsto$
$[m : \text{mem\_obj}, i : \mathbf{N} \mapsto$
$\textit{if}\ \text{resident}(\sigma_i, m, i)\ \textit{then}\ \mu(\sigma_i)(\pi(\sigma_i)(m, \text{align}(i)), \rho(i))\ \textit{else}\ \gamma(\text{vsig}(\sigma_i))(m, i)\,]\,].$

The abstraction function modifies the embedded *vstate* to use γ_{abstr} in place of γ.

Definition 3.8 (abstr) *Theory: vm-impl*
$[\,\sigma_i : \text{istate} \mapsto$
$\quad$ *let* $\sigma : \text{vstate}$ *be* $\text{vsig}(\sigma_i)$ *in*

$$\text{make\_vstate} : \begin{cases} \alpha(\sigma), \\ \beta(\sigma), \\ \Lambda(\sigma), \\ \gamma_{\text{abstr}}(\sigma_i) \end{cases}$$

$].$

3.3 The Operations Supported by the Implementation

The *map* and *unmap* operations remain essentially unchanged in the implementation, and will be omitted from this presentation to save space. The *page-out* and *page-in* operations may occur freely at any time; *page-in* enables *ifetch* and *istore* to occur, as the latter two are possible only if the target address is resident in the store.

The Istore Operation To define *istore*, we define an assignment function for physical store, as well as some auxiliaries to package up address translation.

Definition 3.9 (assign_i) *Theory: vm-impl*
$[\,p : \text{page}, i : \mathbf{N}, w : \text{word}, c : \text{page} \times \mathbf{N} \rightharpoonup \text{word} \mapsto$
$\quad [\,p_1 : \text{page}, j : \mathbf{N} \mapsto$
$\qquad$ *if* $p_1 = p \wedge i = j$ *then* w *else* $c(p_1, j)\,]\,].$

Definition 3.10 (va_to_page) *Theory: vm-impl*
$[\,\sigma_i : \text{istate}, p : \text{process}, s : \text{seg}, o : \text{off} \mapsto$
$\quad$ *let* $\sigma : \text{vstate}$ *be* $\text{vsig}(\sigma_i)$ *in*
$\quad \pi(\sigma_i)(\alpha(\sigma)(p, s), \text{align}(\text{res\_off}(\sigma, p, s, o)))\,].$

Definition 3.11 (va_resident) *Theory: vm-impl*
$[\,\sigma_i : \text{istate}, p : \text{process}, s : \text{seg}, o : \text{off} \mapsto$
$\quad \text{va\_to\_page}(\sigma_i, p, s, o) \downarrow\,].$

Definition 3.12 (va_res_off) *Theory: vm-impl*
$[\,\sigma_i : \text{istate}, p : \text{process}, s : \text{seg}, o : \text{off} \mapsto$
$\quad \rho(\text{res\_off}(\text{vsig}(\sigma_i), p, s, o))\,].$

Definition 3.13 (istore) *Theory: vm-impl*
$[\,\sigma_i : \text{istate}, p : \text{process}, s : \text{seg}, o : \text{off}, v : \text{word} \mapsto$
$\quad \text{make\_istate}(\text{vsig}(\sigma_i),$
$\qquad \pi(\sigma_i),$
$\qquad \text{assign}_i(\text{va\_to\_page}(\sigma_i, p, s, o), \text{va\_res\_off}(\sigma_i, p, s, o), v, \mu(\sigma_i)))\,].$

The first of the main refinement theorems is shown in Figure 2.

Theorem 2. (istore-impl-correct) *Theory: vm-impl*
$\forall \sigma_i$: istate, p : process, s : seg, o : off, v : word
s. t. va_resident(σ_i, p, s, o),
abstr(istore(σ_i, p, s, o, v)) = store(abstr$(\sigma_i), p, s, o, v)$.

Fig. 2. Safety Theorem for *istore* Operation

The Ifetch Operation The implementation function *iref* is conceptually similar to *ref*, but references physical store rather than abstract memory objects. It uses the virtual-to-physical address translations defined above. At this abstraction level, physical store is not viewed as being truly initialized to a null value; rather, *iref* manipulates the way that a user process sees it. Initialized physical store can be justified in a further refinement step using methods like those described in this paper.

Definition 3.14 (iref) *Theory: vm-impl*
[σ_i : istate, p : process, s : seg, o : off $\mapsto$
let pg : page *be* va_to_page(σ_i, p, s, o) *and* $i : \mathbf{N}$ *be* va_res_off(σ_i, p, s, o) *in*
conditionally, if $\mu(\sigma_i)(\text{pg}, i) \downarrow$
- *then* $\mu(\sigma_i)(\text{pg}, i)$
- *else* null_word].

Definition 3.15 (ifetch) *Theory: vm-impl*
[σ_i : istate, p : process, s : seg, o : off, v : word $\mapsto$
conditionally, if $v = \text{iref}(\sigma_i, p, s, o)$
- *then* σ_i
- *else* $\bot$istate].

Theorem 3. (ifetch-impl-correctness) *Theory: vm-impl*
$\forall \sigma_i$: istate, p : process, s : seg, o : off, v : word
s. t. ifetch$(\sigma_i, p, s, o, v) \downarrow$,
abstr(ifetch(σ_i, p, s, o, v)) = fetch(abstr$(\sigma_i), p, s, o, v)$.

Fig. 3. Safety Theorem for *ifetch* Operation

The second main refinement theorem is shown in Figure 3.

The Paging Operations The operation to page in data from a memory object transfers one page's worth of words, starting at a page-aligned address a, from the permanent memory object to physical store. The function *page_at* returns a function that, when applied to a natural number i less than the page size, retrieves the word at location $a + i$. This function determines the behavior of physical store after a page has been brought in.

Definition 3.16 (page_at) *Theory: vm-impl*
$[\,\sigma_i : \text{istate}, m : \text{mem\_obj}, a : \text{aligned} \mapsto$
$\quad [\,i : \mathbf{N} \mapsto$
$\qquad \textit{if}\ i < \text{pagesize}\ \textit{then}\ \gamma(\text{vsig}(\sigma_i))(m, a+i)\ \textit{else}\ \bot\text{word}\,]\,].$

To retrieve a page of data, the target page frame must be free, and the data must not yet be resident. If the former failed, the page table might no longer be injective, while if the latter failed, it might no longer be a functional relation. We have omitted the series of definitions that introduces *page_free* for the sake of space; however,

$$\text{page\_free}(\sigma_i, p) \iff \neg(\exists m : \text{mem\_obj}, a : \text{aligned} \quad \pi(\sigma_i)(m, a) = p).$$

Definition 3.17 (page_in_guard) *Theory: vm-impl*
$[\,\sigma_i : \text{istate}, p : \text{page}, m : \text{mem\_obj}, a : \text{aligned} \mapsto$
conjunction
- $\text{page\_free}(\sigma_i, p)$
- $\neg(\text{resident}(\sigma_i, m, a))\,].$

The *page_in* operation must update the page table π to map the abstract memory to p, and it must update physical store μ at p to reflect the contents of the page's worth of data being transferred.

Definition 3.18 (page_in) *Theory: vm-impl*
$[\,\sigma_i : \text{istate}, p : \text{page}, m : \text{mem\_obj}, a : \text{aligned} \mapsto$
conditionally, if $\text{page\_in\_guard}(\sigma_i, p, m, a)$
- *then* $\text{make\_istate}(\text{vsig}(\sigma_i),$
  $[\,m_1 : \text{mem\_obj}, a_1 : \text{aligned} \mapsto \ \textit{if}\ m_1 = m \wedge a_1 = a\ \textit{then}\ p\ \textit{else}\ \pi(\sigma_i)(m_1, a_1)\,],$
  $[\,p_1 : \text{page}, i : \mathbf{N} \mapsto \ \textit{if}\ p_1 = p\ \textit{then}\ \text{page\_at}(\sigma_i, m, a)(i)\ \textit{else}\ \mu(\sigma_i)(p_1, i)\,])$
- *else* $\bot\text{istate}\,].$

Theorem 4. (page_in-impl-correctness) *Theory: vm-impl*
$\forall \sigma_i : \text{istate}, p : \text{page}, m : \text{mem\_obj}, a : \text{aligned}$
s. t. $\text{page\_in\_guard}(\sigma_i, p, m, a),$
$\text{abstr}(\text{page\_in}(\sigma_i, p, m, a)) = \text{abstr}(\sigma_i).$

Fig. 4. Safety Theorem for *page_in*

The safety theorem for *page_in*, which does not implement any abstract operation, states that it is an abstract no-op (Figure 4).

The *page-out* operation is similar to the *page-in* operation, except that it must instead flush a page's worth of data from physical store back out to disk. The page table is modified to mask out the entry for the newly flushed page. To save space, we will not present the details here.

3.4 Liveness of the Implementation

To state and prove the liveness theorems, we introduce the operating system notion of a paging strategy. We extend the language of *vm-impl* with two function constants. One, *selected_page*, returns the page frame which the operating system would select as target for a *page_in* operation in the current state. It is assumed that this is a free page, whenever there is any free page at all. The other, *rejected_page*, returns the page frame which the operating system would flush during a *page_out* operation in the current state. It is assumed never to return a free page as its value, and to be well-defined whenever the state has a page frame in use. These assumptions are formalized in *vm-impl+*.

Language 3.19 (vm-impl+-language)
Embedded language: *vm-impl*
Constants: selected_page *:* [istate $\rightharpoonup$ page] rejected_page *:* [istate $\rightharpoonup$ page]

Component theory: vm-impl
Top level axioms:

rejected_page-unfree $\forall \sigma_i : \text{istate} \quad \neg(\text{page\_free}(\sigma_i, \text{rejected\_page}(\sigma_i)))$.
rejected_page-defined $\forall \sigma_i : \text{istate} \quad \text{s. t.} \quad \exists p : \text{page} \quad \neg(\text{page\_free}(\sigma_i, p))$,
  $\text{rejected\_page}(\sigma_i) \downarrow$.
selected_page-free $\forall \sigma_i : \text{istate} \quad \text{s. t.} \quad \exists p : \text{page} \quad \text{page\_free}(\sigma_i, p)$,
  $\text{page\_free}(\sigma_i, \text{selected\_page}(\sigma_i))$.

Fig. 5. Components and axioms for vm-impl+

Theory 3.20 (vm-impl+)
Language: *vm-impl+-language*
Component Theories and Axioms: *See Figure 5.*

Definition 3.21 (maybe_flush) *Theory: vm-impl+*
$[\, \sigma_i : \text{istate} \mapsto$
  conditionally, if $\exists p : \text{page} \quad \text{page\_free}(\sigma_i, p)$
  - *then* σ_i
  - *else* $\text{page\_out}(\sigma_i, \text{rejected\_page}(\sigma_i))\,]$.

We give the following definition in β-unreduced form because it is a little more compact and maybe clearer; in the *else* clause, σ_j takes the value $\text{maybe\_flush}(\sigma_i)$.

Definition 3.22 (maybe_page_in) *Theory: vm-impl+*
$[\,\sigma_i : \text{istate}, m : \text{mem\_obj}, a : \text{aligned} \mapsto$
conditionally, if $\text{resident}(\sigma_i, m, a)$
- *then* σ_i
- *else* $[\,\sigma_j : \text{istate} \mapsto \text{page\_in}(\sigma_j, \text{selected\_page}(\sigma_j), m, a)\,](\text{maybe\_flush}(\sigma_i))\,]$.

We now state the last two main refinement theorems, the liveness theorems for *istore* and *ifetch*. Each of them asserts that the operation commutes with *abstr*, after any necessary paging activity has been performed. That is, if σ_j results from σ_i by maybe paging in the required data (which in turn may require flushing a page frame), then

$$\text{abstr}(h(\sigma_j)) \simeq g(\text{abstr}(\sigma_i))$$

where h is the implementation operation (for the chosen parameters), and g is the specification operation (for the same parameters). Again, the $\simeq$ sign here means quasi-equivalence; the formula is true if the left and right hand sides have the same value or lack thereof. Hence, the liveness theorem entails that when the specification operation g can occur, then the implementation operation h can also occur, although possibly after the paging activity. This is the main content added to the safety theorems (Theorems 2 and 3), which are used as lemmas in the proofs of the liveness theorems.

In the local definition of σ_j, the term $\alpha(\text{vsig}(\sigma_i))(p, s)$ selects the permanent memory object that referennces to the virtual address $\langle p, s\rangle$ concern. The term $\text{align}(\text{res\_off}(\text{vsig}(\sigma_i), p, s, o))$ returns the page-aligned address immediately preceding the resolved offset into the object.

Theorem 5. (istore-liveness-theorem) *Theory: vm-impl+*
$\forall \sigma_i : \text{istate}, p : \text{process}, s : \text{seg}, o : \text{off}, v : \text{word}$
let s_j : istate
be $\text{maybe\_page\_in}(\sigma_i, \alpha(\text{vsig}(\sigma_i))(p, s), \text{align}(\text{res\_off}(\text{vsig}(\sigma_i), p, s, o)))$ *in*
$\text{abstr}(\text{istore}(s_j, p, s, o, v)) \simeq \text{store}(\text{abstr}(\sigma_i), p, s, o, v)$.

Theorem 6. (ifetch-liveness-theorem) *Theory: vm-impl+*
$\forall \sigma_i : \text{istate}, p : \text{process}, s : \text{seg}, o : \text{off}, v : \text{word}$
let σ_j : istate
be $\text{maybe\_page\_in}(\sigma_i, \alpha(\text{vsig}(\sigma_i))(p, s), \text{align}(\text{res\_off}(\text{vsig}(\sigma_i), p, s, o)))$ *in*
$\text{abstr}(\text{ifetch}(\sigma_j, p, s, o, v)) \simeq \text{fetch}(\text{abstr}(\sigma_i), p, s, o, v)$.

Fig. 6. Liveness Theorems for *istore* and *ifetch*

3.5 Conclusion

These theorems establish that the implementation is a refinement of the specification. More specifically, every computation of the abstract machine corresponds to a computation of the implementation, and conversely. We have not yet, however, formalized and derived this theorem about computations within IMPS.

The paper was intended to demonstrate a style for specifying and, at least for the initial stages, verifying refinements of a virtual memory system. A realistic verification of a virtual memory system requires many additional ingredients. In particular, issues of concurrency are unavoidably raised; an operating system must schedule a different process when one process takes a page fault and waits for paging to complete. As a consequence, the state at the time that a paging operation completes may differ in other ways from the state at the time it began. Thus, an element of non-determinism arises. In Mach, our primary focus, the multi-threaded character of the kernel ensures that other concurrency control issues also arise.

Nevertheless, state information is indispensable in characterizing the behavior of an operating system, and we believe that these refinement techiques play a crucial role in the state-oriented aspects of providing a reliable, rigorously understood computing system.

Acknowledgment. I am deeply grateful to Bill Farmer and Javier Thayer, my colleagues in developing IMPS.

References

1. W. R. Bevier and L. M. Smith. A mathematical description of the Mach kernel: Entities and relations. Technical Report 88, Computational Logic, Incorporated, Austin, TX, February 1993.
2. J. Davies. Specification and proof in real-time systems. Technical report, Oxford University, Programming Research Group, September 1993. Report to Defence Research Agency, Malvern.
3. W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.
4. W. M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.
5. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.
6. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.
7. W. M. Farmer and F. J. Thayer. Two computer-supported proofs in metric space topology. *Notices of the American Mathematical Society*, 38:1133–1138, 1991.
8. J. D. Guttman and D. M. Johnson. Two applications of formal methods at MITRE. Submitted to FME '94., March 1994.
9. C. A. R. Hoare. Notes on data structuring. In O.-J. Dahl, editor, *Structured Programming.* Academic Press, 1972.

10. Keith Loepere. Mach 3 kernel interfaces. Technical report, Open Software Foundation, Cambridge, MA, July 1992. Jointly copyright by Open Software Foundation and Carnegie-Mellon University.
11. Keith Loepere. Mach 3 server writer's guide. Technical report, Open Software Foundation, Cambridge, MA, July 1992. Jointly copyright by Open Software Foundation and Carnegie-Mellon University.
12. Elliott I. Organick. *The Multics System: An Examination of its Structure.* The MIT Press, Cambridge, MA, 1972.
13. Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

Table of Contents

This article was processed using the LaTeX macro package with LLNCS style