[This Page Intentionally Left Partially Blank.]

[Using FrontMatter Library of 24-Oct-85]

A VERIFIED OPERATING SYSTEM KERNEL

APPROVED BY
DISSERTATION COMMITTEE:

_____

_____

_____

_____

_____

Copyright

by

William R. Bevier

1987

A VERIFIED OPERATING SYSTEM KERNEL

by

WILLIAM R. BEVIER, B.A., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1987

ACKNOWLEDGEMENTS

I want to thank my committee members Bob Boyer, J Moore, J. C. Browne, Don Fussell and Don Good. All contributed to making the dissertation better than I could have done on my own. Particular thanks are due to Don Good, who saw to it that I was supported at the Institute for Computing Science at the University of Texas while working on this dissertation.

Thanks to everyone at the Institute for Computing Science for making it such an enjoyable place to work. Conversations with Matt Kaufmann, Bill Young, Warren Hunt and Mike Smith were very valuable. Matt Kaufmann read the first draft of the dissertation. Larry Smith built a prototype editor-based interface to the Boyer-Moore theorem prover which made my life much easier. Larry Akers and Larry Smith kindly protected me from other duties during the last months of preparation of the dissertation. Thanks again to all.

Thanks to my wife Susan, without whom completion of this dissertation would not have been possible or nearly as satisfying.

William R. Bevier

The University of Texas at Austin
December, 1987

A VERIFIED OPERATING SYSTEM KERNEL

Publication No.

William Richard Bevier, Ph.D.
The University of Texas at Austin, 1987

Supervising Professors: Robert S. Boyer, J Strother Moore

We present a multitasking operating system kernel, called KIT, written in the machine language of a uni-processor von Neumann computer. The kernel is proved to implement, on this shared computer, a fixed number of conceptually distributed communicating processes. In addition to implementing processes, the kernel provides the following verified services: process scheduling, error handling, message passing, and an interface to asynchronous devices. The problem is stated in the Boyer-Moore logic, and the proof is mechanically checked with the Boyer-Moore theorem prover.

Chapter 1

Introduction

1.1 The Thesis

Since Dijkstra's report on the "THE"-multiprogramming system [Dijkstra68], many operating systems have been designed as a hierarchy of cooperating processes. Brinch Hansen [BrinchHansen70] named the lowest layer of such a hierarchy the *nucleus*, or *kernel*. The purpose of a kernel is to simulate processes and implement process communication. The virtual machine which results is a base for building higher layers of an operating system.

To date, research in the verification of operating systems has not adequately penetrated the kernel layer. It is possible to apply formal methods such as Hoare logic to kernel verification, but the specifications which arise are large and tedious to prove. The situation begs for mechanical aids. Some formal systems which can be used to specify operating systems take a notion of *process* as a primitive. Those who attempt to use such systems to verify an implementation typically rely upon low-level machine-dependent procedures which cannot be verified with the formal methods under consideration. These primitive procedures are usually critical to the correct implementation of a process.

The purpose of this work is to address the problem of operating systems kernel verification. In particular, we are concerned with the correct implementation of processes. We present a kernel, which we call "KIT", written in the assembler language of a uni-processor computer with a typical von Neumann architecture. (The name KIT is not a tortured acronym, but is intended to suggest the three words *kernel*, *isolated*, *tasks*.) KIT is proved to implement, on a shared computer, a fixed number of conceptually distributed communicating processes. In addition to implementing processes, the kernel provides the following verified services:

- Process scheduling and allocation of CPU time,
- Response to program error conditions (e.g. unrecognized opcode),
- Single-word message passing among processes,
- Character I/O to asynchronous devices.

The result is an operating system kernel which correctly implements a set of concurrent processes. A set of communicating processes will run as specified on KIT provided there are no hardware errors. The operating system is proved not to introduce implementation bugs. KIT and its specification are defined in the Boyer-Moore logic, and the proof is mechanically checked with the Boyer-Moore theorem prover [ACL].

It is important to say what we do not handle. We take UNIX as a point of comparison. The UNIX kernel as described by Bach [UNIX] contains two main components: the file subsystem, which besides implementing a file structure also hides the device interface from the user level; and the process control subsystem, which includes process creation and deletion, process communication, process scheduling and memory management.

KIT deals with a subset of these phenomena. It handles process scheduling, process communication (by message passing), and a terminal device interface. There is no dynamic creation of processes or communication channels. There is no file system. KIT's memory management is strictly that supplied by the hardware - it does not include virtual memory. The hardware memory management is not assumed to be correct, though. The verification of KIT requires a proof that the hardware protection mechanism permits the implementation of isolated address spaces.

Therefore, while KIT is not big enough to be considered a kernel for a general purpose operating system, it does confront some key operating system phenomena. It is adequate for a small special purpose system such as a communications processor. KIT is fully operational in that it runs on a machine which can be simulated in the Boyer-Moore logic.

1.2 Process Isolation

We identify a process with the machine state to which it has access. Processes have two kinds of state: private and shared. There are two corresponding kinds of transitions on a process: private, which alters only a process's private state, and communication, which may alter the shared state and the private state of a process.

As explained later, the highest level specification for KIT is a definition of a single communicating process, which defines some elementary message passing primitives. These primitives are the only communication transitions available to a process. The remaining transitions are required to be private ones.

Our goal is to prove that multiple instances of the process definition are implemented by KIT running on a particular uni-processor von Neumann machine. In the implementation, a process's private state consists of a segment of machine memory and some CPU registers. Its shared state consists of some message buffers. The private transitions are implemented as the set of non-privileged machine instructions. Communication transitions are implemented as supervisor services implemented by KIT routines.

At the specification level, process isolation is a trivial property. The proof that a task's private state can change only when it is active is a matter of examining a single small definition. At the implementation level, the private state of a process is not transparently isolated from others. It is not at all obvious that a private transition on one process leaves other process states unchanged. To prove that KIT implements multiple processes requires the following results.

- A machine instruction executed in user mode alters only the private state of the current task. The private state of other tasks and the shared state are protected.
- The services implemented by the kernel alter private and shared state only in ways specified by the process definition.

The first result is largely a property of the machine architecture. We prove that

the protection mechanism of the target machine permits the implementation of private state and private transitions as required by the process definition. The kernel guarantees that some conditions required by the machine protection theorem are invariant.

The second result is obtained by verifying kernel code. We prove that state changes made by each kernel routine correspond to changes to the abstract state at the level of the process definition.

1.3 A Characterization of this Work

We wish to leave the reader with no doubt regarding one of our goals: to verify KIT at the machine code level. Below we give a small portion of an assembler language listing of the kernel. This is the portion which saves the state of the current task on an interrupt. We give the details of our implementation of KIT in chapter IMPLEMENTATION, but we hope at this point to emphasize the level of verification we perform. This code places the address of an entry in a task table into register 2, and saves the task-visible state of the CPU of our target machine in that entry. Our verification proves that the processor state is saved correctly so that the fiction that each process owns the processor is maintained.

The assembler language representation of our code is not the object of proof. We go yet lower. We verify the assembler output of the source code: a sequence of numbers which our target machine is capable of interpreting.

```
SAVE-STATE
(move  (2 temp-r2) (1 r2))             ;; Save R2
(move  (2 temp-r3) (1 r3))             ;; Save R3
(move  (1 r3) readyq)                  ;; R3 points to ready queue
(call  qfirst)                         ;; R2 has current task id
save-state-return
(mult  (1 r2) task-table-entry-length) ;; multiply by task table entry length
(add   (1 r2) task-table)              ;; R2 points to current task table entry
(move  (3 r2 pc-field)   (2 reg-save-area interrupt-pc-field))
(move  (3 r2 sp-field)   (2 reg-save-area interrupt-sp-field))
(move  (3 r2 r2-field)   (2 temp-r2))
(move  (3 r2 r3-field)   (2 temp-r3))
(move  (3 r2 r4-field)   (1 r4))
(move  (3 r2 r5-field)   (1 r5))
(move  (3 r2 r6-field)   (1 r6))
(move  (3 r2 r7-field)   (1 r7))
(add   (1 r2) flag-field)              ;; bump index register
(move  (3 r2)            (2 reg-save-area interrupt-flag-field))
(move  (1 r2)            (2 temp-r2))  ;; Restore R2 & R3.
(move  (1 r3)            (2 temp-r3))  ;; This is necessary for SVC interrupts.
(return)
```

1.4 Plan of Dissertation

The script of Boyer-Moore forms which defines and verifies KIT contains approximately one thousand definitions and thirty-five hundred lemmas. This script is the heart of the dissertation. The challenge is to explain this script in a coherent fashion. In Chapter INTERPRETERS we discuss our approach to verifying KIT. Chapters SPECIFICATION through VERIFICATION discuss the specification, implementation and verification of KIT by examining the highest level definitions and theorems in the script. Chapter QUEUES presents in detail the specification, implementation and verification of queues, which permeate the kernel. Chapter CONCLUSION surveys related work and summarizes our effort. Subsequent volumes contain the script and an index of names in the script. Numbers printed with events in the text are indices into the script. The index gives the page numbers on which events occur in this volume.

1.5 The Boyer-Moore Logic and its Proof Checker

A description of the Boyer-Moore logic and its proof checker appears in Appendix BMLOGIC. The description is taken with permission from [quant]. We make some comments on our usage of the theorem prover. These comments assume familiarity with the logic and the theorem prover.

We make use of shells to define a number of record structures. We describe our shells by giving the shell constructor, shell recognizer and shell accessors as shown in the example of the shell **FOO** below. This example illustrates how we display an **ADD-SHELL** event.

```
Shell Definition.
Add the shell FOO with recognizer FOO-SHELLP,
defining the record structure <A, B>.
```

We place no type restrictions on the fields of a shell. The event **ADD-SHELL** in the Boyer-Moore logic does not permit associating arbitrary predicates with fields. Since we cannot say everything about a shell within the **ADD-SHELL** form, we choose to say nothing. If we wish to restrict the fields of a shell to have certain values, we define a predicate in the logic which recognizes a constrained shell.

We found that we could not manage our large script of events with globally enabled rewrite rules and definitions. Each event in the script is therefore immediately disabled. The **DISABLE** events are not displayed in the script but should be understood to be present. Our approach to guiding the theorem prover to a proof therefore requires liberal use of **ENABLE** hints on lemmas. We found this actually to be quite congenial. A given lemma typically immediately relies upon a fairly small number of support lemmas and definitions. When proposing a lemma to the theorem prover, we can guess at a number of definitions and lemmas which must be enabled. The others we discover as we see the prover fail. We found that using this approach we were always engaged in a positive proof search and were never battling a rewriter which was taking us in a bad direction due to an enabled but forgotten rule. As we became more and more familiar with our script we found we were able to remember the names of many lemmas. We also invented mechanical aids for discovering the names of applicable lemmas.

While a lemma *typically* relies on a small number of immediate supporters, there are exceptions. To ease the burden of enabling large numbers of events, we created a new event **DEFTHEORY**. The form **(DEFTHEORY <NAME> <LIST-OF-NAMES>)** binds a name to a list of earlier event names. In subsequent events, the hint **(ENABLE-THEORY NAME)** enables all events to which **NAME** is bound.

Chapter 2

Defining Finite State Machines with Recursive Functions

KIT is verified by proving a correspondence between the behavior of two finite state machines. An abstract finite state machine serves as an operational specification. The kernel running on the bare computer is also defined as a finite state machine. In this chapter we explain how we define finite state machines, and describe the form of the correspondence theorem between two machines. We give a brief overview of the kernel proof, stating the correspondence theorem which establishes KIT's correctness.

2.1 Interpreters

We define a finite state machine by an *interpreter function*. An interpreter function models transitions to a machine over an arbitrary but finite time span. It is a dyadic function of the form $Int : S \times O \rightarrow S$, where S is a set of machine states and O is a set of oracles for a machine. An oracle has two roles. It determines the finite time span for which a machine invocation operates, and it may introduce non-deterministic state changes into a machine, including communication with other machines.

In a simple situation the set of natural numbers N can be chosen as the oracle set. An interpreter of the form $Int : S \times N \rightarrow S$ models a machine which operates in complete isolation. Such a machine can be defined in the Boyer-Moore logic as follows. The function **STEP** advances the state of this machine. The expression **(MACHINE1 STATE N)** is the state obtained by applying **N** successive applications of **STEP** to **STATE**.

```
Definition.
(MACHINE1 STATE N)
   =
(IF (ZEROP N)
    STATE
    (MACHINE1 (STEP STATE) (SUB1 N)))
```

In a more typical situation, an oracle is a list which represents a finite time-sequenced series of external events impinging on a machine. The length of the oracle determines the time span over which the machine operates. An element of the oracle is either a single external event, or a symbol such as **'TICK** indicating no event. The interpreter consumes the next element of the oracle at each step, and runs until the oracle is exhausted. The definition of **MACHINE2** gives the form of such an interpreter. In this example, the function **CONSUME-INPUT** consumes the next element of the oracle, incorporating it into the state of the machine so that the input is visible to **STEP**.

```
Definition.
(MACHINE2 STATE ORACLE)
   =
(IF (NOT (LISTP ORACLE))
    STATE
    (MACHINE2 (STEP (CONSUME-INPUT STATE (CAR ORACLE)))
              (CDR ORACLE)))
```

2.2 Interpreter Equivalence Theorems

In this section we describe several types of theorem which establish a correspondence between two machines. We call such theorems *interpreter equivalence theorems*.

We wish to define an *implements* relation on two machines. Let $Int_A : S_A \times O_A \rightarrow S_A$ and $Int_C : S_C \times O_C \rightarrow S_C$ be interpreter functions which define two machines M_A and M_C. (The subscripts A and C are chosen to suggest *abstract* and *concrete* machines.) Let $MapUp : S_C \rightarrow S_A$ be an abstraction function which maps a concrete state to an abstract state. We say that M_C *implements* M_A if the following theorem holds.

$$\forall\, s_C \in S_C$$
$$\forall\, o_A \in O_A$$
$$(1) \qquad \exists\, o_C \in O_C \text{ such that}$$
$$MapUp\,(Int_C\,(s_C, o_C)) = Int_A\,(MapUp\,(s_C), o_A)\,.$$

Figure INTERPRETER-EQUIVALENCE illustrates the correspondence which the *implements* relation establishes.

Figure 2-1: Interpreter Equivalence

In this paper we prove a theorem of the form of *(1)*. Notice that if there is a function *MapDown* : $S_A \rightarrow S_C$, and $\forall\ s_A \in S_A$, *MapUp (MapDown* $(s_A)) = s_A$, then from *(1)* we get a stronger relation given by *(2)*.

$\forall\ s_A \in S_A$
$\forall\ o_A \in O_A$
(2) $\exists\ o_C \in O_C$ *such that*

MapUp $(Int_C$ *(MapDown* (s_A), $o_C)) = Int_A\ (s_A, o_A)$ **.**

Sometimes we find it convenient to reverse the quantification on the abstract and concrete oracles. Then we get an interpreter equivalence theorem of the form given by *(3)*. Figure INTERPRETER-EQUIVALENCE also describes this formula.

$\forall\ s_C \in S_C$
$\forall\ o_C \in O_C$
(3) $\exists\ o_A \in O_A$ *such that*

MapUp $(Int_C\ (s_C, o_C)) = Int_A$ *(MapUp* (s_C), $o_A)$

We cannot state *(1)*, *(2)* or *(3)* in the quantifier-free Boyer-Moore logic. For *(1)* we replace the existential variable o_C with a function **CORACLE** which computes the oracle required by Int_C to match the behavior of Int_A. Typically, this is a function both of the initial concrete state and the value of o_A. We re-state *(1)* in the Boyer-Moore logic as follows. The predicate **GOOD-CSTATE** identifies an element of the set of concrete machine states.

```
Theorem.  IMPLEMENTS-RELATION:
(IMPLIES (GOOD-CSTATE CSTATE)
         (EQUAL (MAPUP (INT-C CSTATE (CORACLE CSTATE ORACLE)))
                (INT-A (MAPUP CSTATE) ORACLE)))
```

2.3 The KIT Proof Structure

The main result in the verification of KIT is the theorem **OS-IMPLEMENTS-PARALLEL-TASKS**. It is an interpreter equivalence theorem which demonstrates that the behavior of a single task running under the kernel implements an abstract definition of a process. In this theorem, the functions **TASK-PROCESSOR** and **TM-PROCESSOR** are interpreter functions. The function **PROJECT-ITH-TASK** is the mapping function. Our goal in this dissertation is to explain the content of this theorem.

```
Theorem {4623}. OS-IMPLEMENTS-PARALLEL-TASKS:
(IMPLIES
 (AND (GOOD-OS OS)
      (PLISTP ORACLE)
      (FINITE-NUMBERP I (LENGTH (AK-PSTATES (MAPUP-OS OS)))))
 (EQUAL (PROJECT-ITH-TASK I (TM-PROCESSOR OS (OS-ORACLE OS ORACLE)))
        (TASK-PROCESSOR (PROJECT-ITH-TASK I OS)
                        I
                        (CONTROL-ORACLE I (MAPUP-OS OS) ORACLE))))
```

The problem is decomposed into two steps, as pictured in Figure PROOF-STRUCTURE. An intermediate machine, called the *abstract kernel* gives an operational specification for KIT. The proof of **OS-IMPLEMENTS-PARALLEL-TASKS** is a result of the theorems **AK-IMPLEMENTS-PARALLEL-TASKS** and **CORRECTNESS-OF-OPERATING-SYSTEM**, which handle the top and bottom interpreter equivalence theorems, respectively, of Figure PROOF-STRUCTURE.

Figure 2-2: KIT Proof Structure

```
Theorem {1689}.  AK-IMPLEMENTS-PARALLEL-TASKS (rewrite):
(IMPLIES (AND (GOOD-AK AK)
              (FINITE-NUMBERP I (LENGTH (AK-PSTATES AK))))
         (EQUAL (PROJECT I (AK-PROCESSOR AK ORACLE))
                (TASK-PROCESSOR (PROJECT I AK)
                                I
                                (CONTROL-ORACLE I AK ORACLE))))

Theorem {4621}.  CORRECTNESS-OF-OPERATING-SYSTEM (rewrite):
(IMPLIES (AND (GOOD-OS OS)
              (PLISTP ORACLE))
         (EQUAL (MAPUP-OS (TM-PROCESSOR OS (OS-ORACLE OS ORACLE)))
                (AK-PROCESSOR (MAPUP-OS OS) ORACLE)))
```

The verification of KIT spans these layers of interpreters. The *task* layer is at the top. It provides a definition of a single communicating process. The second layer, the *abstract kernel*, gives the kernel specification. The abstract kernel contains a fixed number of task states. The state space of the abstract kernel is such that the isolation of task states is easily established. A function **PROJECT** maps the state of *i*th task out of the abstract kernel and up to the task layer.

The bottom layer defines the target machine. The target machine is a very simple von Neumann computer. We are particularly interested in the state of a target machine when loaded with the machine code for KIT. In such a machine state, defined by the predicate **GOOD-OS**, the implementations of tasks are not transparently isolated. We must prove that they are isolated as defined by the abstract kernel. The function **MAPUP-OS** maps the kernel state up to an abstract kernel state. It not only maps up the state of each task, but the state of all data structures (e.g. the ready queue) which the kernel uses to manage tasks.

Chapter 3

The Specification of KIT

In this chapter we describe the finite state machines which define the task and abstract kernel layers of Figure 2-2. These serve as specifications for KIT. For each layer we describe a state set and an interpreter function. We occasionally make reference to intended implementation details to foreshadow later chapters.

3.1 The Task Layer

The top layer defines an independent process, called a *task*, capable of communicating with other processes. We wish to prove correct a particular implementation of tasks.

Figure NETWORK depicts an instance of a network structure of communicating processes. This figure contains a star with five points, while our definition allows an arbitrary but fixed number of points. Single-headed arrows indicate communication in the direction of the arrowhead. Double-headed arrows abbreviate two single-headed arrows, one going in each direction. Each node of Figure NETWORK represents a process. The nodes at the points of the star are implemented as KIT tasks. The nodes at the extreme perimeter, which communicate with tasks in one direction only, are implemented as I/O devices.

The task layer defines a single task's view of this process network. The state space of a task consists of two parts: a private state which is accessible only to the owning task, and a shared state which is used for implementing inter-task communication. We distinguish two categories of transitions on a task: private transitions update only the private state, communication transitions update the shared state. The

Figure 3-1: Network

state space of a task is described in the Boyer-Moore logic by the shell **TASK**. The **TASK-PSTATE** field is the private state of a task. The **TASK-CHANNELS** field is the shared state containing an implementation of the communication network in which tasks participate.

```
Shell Definition {1386}.
Add the shell TASK with recognizer TASK-SHELLP,
defining the record structure <TASK-PSTATE, TASK-CHANNELS>.
```

We remind readers unfamiliar with the Boyer-Moore logic that the form **(TASK A B)** constructs a task state with private state **A** and channel state **B**. If **X** is a task object, then the form **(TASK-PSTATE X)** accesses its private state field, and **(TASK-CHANNELS X)** accesses its channel state.

The **TASK-CHANNELS** field contains an implementation of the network structure. It is a three-tuple of tables of fixed-size buffers. The **TASK-IBUFFERS** table is for communication with input devices, the **TASK-OBUFFERS** table is for communication with output devices, and the **TASK-MBUFFERS** table contains message buffers for communicating with other tasks. The names we place on these fields merely suggest a lower-level implementation. At this level, a task's view of a device differs from its view of another task only in the name space each occupies, as suggested by Figure 3-1.

```
Definition {1387}.
(TASK-IBUFFERS TASK) = (CAR (TASK-CHANNELS TASK))

Definition {1388}.
(TASK-OBUFFERS TASK) = (CADR (TASK-CHANNELS TASK))

Definition {1389}.
(TASK-MBUFFERS TASK) = (CADDR (TASK-CHANNELS TASK))
```

The predicate **GOOD-TASK** completes the definition of the state set of a task. It recognizes a proper task state with given limits on the number of buffers. The predicates **GOOD-TASK-BUFFER-LIST** and **GOOD-TASK-BUFFER-TABLE** place limits on the length of buffers and the type of their contents. The predicate **GOOD-ADDRESS-SPACE** recognizes a proper target machine address space. It reveals our intention to implement the private state of a task as an address space of some target machine. At this point, we offer no definition of **GOOD-ADDRESS-SPACE.**

```
Definition {1433}.
(GOOD-TASK TASK ILENGTH OLENGTH MLENGTH)
   =
(AND (TASK-SHELLP TASK)
     (GOOD-ADDRESS-SPACE (TASK-PSTATE TASK)
                         (LENGTH (TM-MEMORY (TASK-PSTATE TASK))))
     (EQUAL (LENGTH (TASK-IBUFFERS TASK)) ILENGTH)
     (GOOD-TASK-BUFFER-LIST (TASK-IBUFFERS TASK)
                            (TASK-IBUFFER-CAPACITY))
     (EQUAL (LENGTH (TASK-OBUFFERS TASK)) OLENGTH)
     (GOOD-TASK-BUFFER-LIST (TASK-OBUFFERS TASK)
                            (TASK-OBUFFER-CAPACITY))
     (EQUAL (LENGTH (TASK-MBUFFERS TASK)) MLENGTH)
     (GOOD-TASK-BUFFER-TABLE (TASK-MBUFFERS TASK)
                             MLENGTH
                             (TASK-MBUFFER-CAPACITY)))
```

The interpreter function which defines the transitions on a task is called **TASK-PROCESSOR**. The first formal argument, **TASK**, is a task state. For convenience, and this is the only place we diverge from the pattern, we split this interpreter's oracle into two

formal arguments. The argument **I** is the identifier of the task in the network which the task can sense only through its shared state. The task identifier is a non-negative integer in some bounded range. The argument **ORACLE** is a list each of whose elements is either **T**, indicating that the task is active and should take a step on its own initiative, or not **T**, indicating that the task is not active at this step. In the latter case, the oracle supplies a triple which contains the value of the channel state at the end of the current step. We shall see later that the kernel, which implements a fixed number of task states, can construct the oracle argument to a task. Notice that the function **TASK-UPDATE-CHANNELS**, which updates a task state on a non-active step, preserves the private state of the task. Therefore a task's private state is not altered when the task is not active.

An active task step is defined by the function **TASK-STEP**. The predicate **TASK-COMMUNICATIONP** determines if the current transition is a communication transition. If so, the task executes a communication step, otherwise a private step. A private step is defined to be a fetch-execute operation. We thus require a task's private state to contain its own control state. There is no requirement in this definition that only a single task is active in any instant, but KIT runs on a single processor and implements tasks in this way.

```
Definition {1425}.
(TASK-PROCESSOR TASK I ORACLE)
   =
(IF (LISTP ORACLE)
    (IF (TASK-ACTIVEP (CAR ORACLE))
        (TASK-PROCESSOR (TASK-STEP TASK I)
                        I
                        (CDR ORACLE))
        (TASK-PROCESSOR (TASK-UPDATE-CHANNELS TASK (CAR ORACLE))
                        I
                        (CDR ORACLE)))
    TASK)

Definition {1424}.
(TASK-ACTIVEP X) = (EQUAL X T)

Definition {1422}.
(TASK-STEP TASK I)
   =
(IF (TASK-COMMUNICATIONP TASK)
    (TASK-COMMUNICATION-STEP TASK I)
    (TASK-PRIVATE-STEP TASK))

Definition {1423}.
(TASK-UPDATE-CHANNELS TASK CHANNELS)
   =
(TASK (TASK-PSTATE TASK) CHANNELS)
```

```
Definition {1421}.
(TASK-PRIVATE-STEP TASK)
   =
(TASK (TASK-FETCH-EXECUTE (TASK-PSTATE TASK))
      (TASK-CHANNELS TASK))
```

The definition of **TASK-COMMUNICATION-STEP** specifies the communication primitives which the kernel implements. These are the operations *send*, *receive*, *input* and *output*. *Send* and *receive* access the message buffers, *input* the input buffers and *output* the output buffers. There is one bounded message buffer for each <*i*,*j*> pair of task identifiers. Message buffer <*i*,*j*> handles messages flowing from task *i* to task *j*. Communication with input and output buffers is simpler. Task *i* can receive only from input buffer *i*, and can send only to output buffer *i*. The units of information which are passed are implemented as single target machine words.

The communication primitives are sensitive to empty and full buffers. An attempt to retrieve information from an empty buffer results in no change to the task state, so the next time the task is active it will be in the same state from which it initially tried to receive and will therefore attempt to retrieve from the same buffer again. We give the definitions of *send*, *receive*, *input* and *output* below. The function **TASK-STORE-MESSAGE** defines a convention by which messages are delivered to the private state of a task. The function **TASK-UPDATE-CONTROL** updates the control state of a task so that the communication operation is stepped over.

```
Definition {1416}.
(TASK-EXECUTE-SEND MSG SRCID DESTID TASK)
   =
(IF (QFULLP2 SRCID DESTID (TASK-MBUFFERS TASK) (TASK-MBUFFER-CAPACITY))
    TASK
    (TASK (TASK-UPDATE-CONTROL (TASK-PSTATE TASK))
          (LIST (TASK-IBUFFERS TASK)
                (TASK-OBUFFERS TASK)
                (ENQ2 MSG SRCID DESTID (TASK-MBUFFERS TASK)))))
```

```
Definition {1417}.
(TASK-EXECUTE-RECEIVE SRCID DESTID TASK)
   =
(IF (QEMPTYP2 SRCID DESTID (TASK-MBUFFERS TASK))
    TASK
    (TASK (TASK-UPDATE-CONTROL
            (TASK-STORE-MESSAGE
              (QFIRST2 SRCID DESTID (TASK-MBUFFERS TASK))
              (TASK-PSTATE TASK)))
          (LIST (TASK-IBUFFERS TASK)
                (TASK-OBUFFERS TASK)
                (DEQ2 SRCID DESTID (TASK-MBUFFERS TASK)))))

Definition {1418}.
(TASK-EXECUTE-OUTPUT CHAR ID TASK)
   =
(IF (QFULLP (GETNTH ID (TASK-OBUFFERS TASK)) (TASK-OBUFFER-CAPACITY))
    TASK
    (TASK (TASK-UPDATE-CONTROL (TASK-PSTATE TASK))
          (LIST (TASK-IBUFFERS TASK)
                (ENQ-ITH-BUFFER CHAR ID (TASK-OBUFFERS TASK))
                (TASK-MBUFFERS TASK))))

Definition {1419}.
(TASK-EXECUTE-INPUT ID TASK)
   =
(IF (QEMPTYP (GETNTH ID (TASK-IBUFFERS TASK)))
    TASK
    (TASK (TASK-UPDATE-CONTROL
            (TASK-STORE-MESSAGE
              (QFIRST (GETNTH ID (TASK-IBUFFERS TASK)))
              (TASK-PSTATE TASK)))
          (LIST (DEQ-ITH-BUFFER ID (TASK-IBUFFERS TASK))
                (TASK-OBUFFERS TASK)
                (TASK-MBUFFERS TASK))))
```

The functions **GETNTH** and **PUTNTH** are the list accessing primitives. **GETNTH** accesses the *n*th element of a list. **PUTNTH** stores a value in the *n*th location in a list.

```
Definition {210}.
(GETNTH N L)
   =
(IF (LISTP L)
    (IF (ZEROP N)
        (CAR L)
        (GETNTH (SUB1 N) (CDR L)))
    0)

Definition {211}.
(PUTNTH V N L)
   =
(IF (LISTP L)
    (IF (ZEROP N)
        (CONS V (CDR L))
        (CONS (CAR L)
              (PUTNTH V (SUB1 N) (CDR L))))
    L)
```

A list structure is used to represent buffers. Buffers are bounded FIFO queues. The primitives which manipulate a buffer are given below. They are all obvious, except perhaps **QREPLACE**, which replaces the last element of a queue with a new item. The functions **ENQ2**, **DEQ2**, **QFIRST2**, **QFULLP2** and **QEMPTYP2** mentioned above access a 2-dimensional table of buffers, and are defined in terms of the primitives listed below.

```
Definition {470}.
(QFIRST LIST) = (CAR LIST)

Definition {471}.
(ENQ ITEM LIST) = (APPEND LIST (LIST ITEM))

Definition {472}.
(DEQ LIST) = (CDR LIST)

Definition {473}.
(QEMPTYP LIST) = (EQUAL (LENGTH LIST) 0)

Definition {474}.
(QFULLP LIST MAX) = (NOT (LESSP (LENGTH LIST) MAX))

Definition {475}.
(QREPLACE ITEM QUEUE) = (ENQ ITEM (NONLAST QUEUE))
```

The communication primitives are the only transitions explicitly defined at the task layer. Recall that the definition of a private step is the application of a fetch-execute operation to the private state of a task. We intend to define **TASK-FETCH-EXECUTE** to be exactly a target machine's fetch-execute operation. The verification of KIT includes a proof that the target machine's architecture implements isolated address spaces in a way which satisfies this definition of a task.

```
Definition {1415}.
(TASK-FETCH-EXECUTE PSTATE) = (TM-FETCH-EXECUTE PSTATE)
```

This concludes the description of our definition of a task. We have in mind a network of communicating processes whose communication structure is suggested by Figure 3-1. The task layer formalizes the view of this network taken by one of the nodes at the points of the star. We intend to implement the network on a computer running a multi-programmed operating system connected to a set of asynchronous input and output devices. Figure 3-1 suggests clearly our intended implementation.

- *Tasks* are those processes which communicate via the message buffers. A full star network among tasks is defined. They are completely implemented by an operating system running on a computer.
- *Input devices* communicate only with tasks, and in one direction only: device to task.

- *Output devices* communicate only with tasks, and in one direction only: task to device.

The task layer serves as a specification for the kernel. The channel state and channel transitions are completely defined. Private state and private transitions are defined to coincide with some implementation machine. Choosing the private state to be implemented as an address space on a target machine is an idea common in operating systems. The proof that our chosen target machine implements private states in a way which satisfies our definition of a task is one of the most important results in the verification of KIT.

3.2 The Abstract Kernel Layer

The task layer defines the communication transitions in which a task may engage, but says nothing of how tasks are activated. The abstract kernel layer defines a scheme for activating a finite set of tasks. The distinction between a task and an I/O device is made more concrete. Each task has a state known completely to the abstract kernel, while the state of an I/O device is unspecified. Devices communicate with the kernel only through shared ports. A number of task management operations are specified, including time slicing, scheduling and error handling.

The state space of the abstract kernel is described by the shell **AK** which defines a 10-tuple. The **AK-PSTATES** field is a fixed-size array of the private states of tasks. The private state of a task is easily proved to be isolated from the others by virtue of the properties of array access. The fields **AK-IBUFFERS**, **AK-OBUFFERS** and **AK-MBUFFERS** contain the shared state and, when grouped into a list, are identical to the channel state at the task layer. The remaining fields introduce the state required to implement task management and communication with I/O devices. The **AK-READYQ** is a queue of task identifiers. Task identifiers are integers in a range bounded by the number of tasks. The first element of the ready queue is the identifier of the current task. The field **AK-STATUS** is an array, one element for each task, which gives the current status of the task. The **AK-RWSTATE** field is a running/wait state flag. The kernel waits when no tasks are ready to run. The field

AK-CLOCK is the program timer used to control time slicing. The fields **AK-IPORTS** and **AK-OPORTS** define an array of input and output ports for communication with devices.

```
Shell Definition {1443}.
Add the shell AK with recognizer AK-SHELLP,
defining the record structure
<AK-PSTATES, AK-IBUFFERS, AK-OBUFFERS, AK-MBUFFERS, AK-READYQ,
 AK-STATUS, AK-RWSTATE, AK-CLOCK, AK-IPORTS, AK-OPORTS>.
```

The predicate **GOOD-AK** defines the abstract kernel state set. It places restrictions on each field of an **AK** shell. In addition, **GOOD-AK** states two invariants on the abstract kernel. First, the ready queue is a permutation of the set of ready tasks as defined by the task status array. Second, the kernel is in the wait state if and only if the ready queue is empty. These two invariants are required to prove that the predicate **GOOD-AK** is an invariant on the abstract kernel interpreter. The constant function **AK-TASKIDLUB** defines the number of tasks which **AK** supports.

```
Definition {1444}.
(AK-TASKIDLUB) = 16

Definition {1532}.
(GOOD-AK AK)
   =
(AND (AK-SHELLP AK)
     (EQUAL (LENGTH (AK-PSTATES AK)) (AK-TASKIDLUB))
     (GOOD-ADDRESS-SPACE-LIST (AK-PSTATES AK))
     (EQUAL (LENGTH (AK-IBUFFERS AK)) (AK-TASKIDLUB))
     (GOOD-TASK-BUFFER-LIST (AK-IBUFFERS AK) (TASK-IBUFFER-CAPACITY))
     (EQUAL (LENGTH (AK-OBUFFERS AK)) (AK-TASKIDLUB))
     (GOOD-TASK-BUFFER-LIST (AK-OBUFFERS AK) (TASK-OBUFFER-CAPACITY))
     (EQUAL (LENGTH (AK-MBUFFERS AK)) (AK-TASKIDLUB))
     (GOOD-TASK-BUFFER-TABLE (AK-MBUFFERS AK)
                             (AK-TASKIDLUB)
                             (TASK-MBUFFER-CAPACITY))
     (PLISTP (AK-READYQ AK))
     (LESSP (LENGTH (AK-READYQ AK)) (ADD1 (AK-TASKIDLUB)))
     (FINITE-NUMBER-LISTP (AK-READYQ AK) (AK-TASKIDLUB))
     (EQUAL (LENGTH (AK-STATUS AK)) (AK-TASKIDLUB))
     (GOOD-STATUS-LIST (AK-STATUS AK))
     (FINITE-NUMBERP (AK-RWSTATE AK) 2)
     (FINITE-NUMBERP (AK-CLOCK AK) (TM-WORDLUB))
     (PLISTP (AK-IPORTS AK))
     (EQUAL (LENGTH (AK-IPORTS AK)) (TM-PORT-LENGTH))
     (GOOD-TM-IPORT-ARRAY (AK-IPORTS AK))
     (PLISTP (AK-OPORTS AK))
     (EQUAL (LENGTH (AK-OPORTS AK)) (TM-PORT-LENGTH))
     (GOOD-TM-OPORT-ARRAY (AK-OPORTS AK))
     (PERMUTATION (AK-READYQ AK) (AK-READY-SET AK))
     (IFF (AK-WAITING AK) (QEMPTYP (AK-READYQ AK))))

Definition {357}.
(FINITE-NUMBERP N LUB) = (AND (NUMBERP N) (LESSP N LUB))
```

```
Definition {359}.
(FINITE-NUMBER-LISTP L LUB)
   =
(IF (LISTP L)
    (AND (FINITE-NUMBERP (CAR L) LUB)
         (FINITE-NUMBER-LISTP (CDR L) LUB))
    T)
```

The interpreter function which defines the transitions on the abstract kernel is **AK-PROCESSOR**. The argument **AK** represents the state of the abstract kernel. The oracle argument is a list. Each element of the list is either an input interrupt, an output interrupt or neither. An input interrupt is a 2-tuple containing a device identifier and a character. An output interrupt is a 1-tuple containing only a device identifier. The function **AK-POST-INTERRUPT** incorporates an interrupt into the state of the machine by updating one of the ports. **AK-POST-INTERRUPT** raises the interrupt flag in an input port on an input interrupt, and writes the character into a character buffer in the port. Similarly, **AK-POST-INTERRUPT** raises an interrupt in an output port on an output interrupt. When an oracle element is not an I/O interrupt, no state change is made by **AK-POST-INTERRUPT**. The abstract kernel is defined to post interrupts in a way identical to the target machine. Chapter IMPLEMENTATION contains the formal details about the structure of ports and I/O interrupt posting.

```
Definition {1516}.
(AK-PROCESSOR AK ORACLE)
   =
(IF (LISTP ORACLE)
    (AK-PROCESSOR (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                  (CDR ORACLE))
    AK)
```

The function **AK-STEP** defines the single-step function of the abstract kernel. Input and output interrupt processing has the highest priority. The functions **AK-INPUT-INTERRUPT-HANDLER** and **AK-OUTPUT-INTERRUPT-HANDLER** define the input and output interrupt handlers. **AK-WAITING** determines if the machine is in the wait state. If so, no state change occurs. If none of the above conditions hold, the error status of the current task is checked. The function **AK-ERROR-HANDLER** defines the kernel's error handler. A clock interrupt signals the end of the current task's time slice. The function **AK-CLOCK-INTERRUPT-HANDLER** defines the task switch on a clock interrupt. The function

AK-SVC-INTERRUPTP detects a request to call a kernel function in behalf of the current task ("svc" abbreviates "supervisor call"). The services provided by the kernel are exactly the communication primitives of the task layer: *send*, *receive*, *input* and *output*. The function **AK-SVC-HANDLER** defines these operations at the abstract kernel layer. Finally, if none of the above conditions hold, the current task takes a private step as defined by **AK-PRIVATE-STEP**.

Like the private step function at the task layer, **AK-PRIVATE-STEP** depends on the target machine's fetch-execute function, **TM-FETCH-EXECUTE**. **AK-PRIVATE-STEP** applies **TM-FETCH-EXECUTE** to the current task's private state. More precisely, the *i*th element of the private state array is replaced by the application of **TM-FETCH-EXECUTE** to that element, where *i* is the identifier of the current task. The isolation of private states is a simple result.

```
Definition {1514}.
(AK-STEP AK)
   =
(IF (AK-INPUT-INTERRUPTP AK)
    (AK-INPUT-INTERRUPT-HANDLER
      (AK-INTERRUPTING-INPUT-PORT (AK-IPORTS AK))
      AK)

(IF (AK-OUTPUT-INTERRUPTP AK)
    (AK-OUTPUT-INTERRUPT-HANDLER
      (AK-INTERRUPTING-OUTPUT-PORT (AK-OPORTS AK))
      AK)

(IF (AK-WAITING AK)
    AK

(IF (AK-ERRORP AK)
    (AK-ERROR-HANDLER AK)

(IF (AK-CLOCK-INTERRUPTP AK)
    (AK-CLOCK-INTERRUPT-HANDLER AK)

(IF (AK-SVC-INTERRUPTP AK)
    (AK-SVC-HANDLER AK)

    (AK-PRIVATE-STEP AK)))))))
```

```
Definition {1505}.
(AK-PRIVATE-STEP AK)
   =
(AK (AK-FETCH-EXECUTE (AK-TASKID AK) (AK-PSTATES AK))
    (AK-IBUFFERS AK)
    (AK-OBUFFERS AK)
    (AK-MBUFFERS AK)
    (AK-READYQ AK)
    (AK-STATUS AK)
    (AK-RWSTATE AK)
    (SUB1 (AK-CLOCK AK))
    (AK-IPORTS AK)
    (AK-OPORTS AK))

Definition {1461}.
(AK-FETCH-EXECUTE ID PSTATES)
   =
(PUTNTH (TM-FETCH-EXECUTE (GETNTH ID PSTATES))
        ID
        PSTATES)

Definition {1446}.
(AK-TASKID AK) = (QFIRST (AK-READYQ AK))
```

An **AK** step is an application of one of five interrupt functions, or is a private step, or is a noop in the case of a waiting machine with no I/O interrupts. The definitions of the five **AK** interrupt handlers provide a specification for the services which must be provided by the implementation of KIT on the target machine. The definition of a private step establishes a constraint on the protection mechanism provided by the target machine's architecture. In the remainder of this section we examine each of the five interrupt handlers, beginning with the simplest.

3.2.1 The Clock Interrupt Handler

AK-CLOCK-INTERRUPT-HANDLER defines a simple round-robin scheduling algorithm. The identifier of the current task is the first element of the ready queue. On a clock interrupt, the first element of the ready queue is removed and enqueued at the end of the ready queue. The dispatcher senses an empty ready queue and sets the kernel state accordingly: the kernel is put in the wait state if the ready queue is empty, otherwise the kernel is put in the run state and the program clock is initialized. On a clock interrupt the length of the ready queue is not changed, so the former condition does not hold. The same primitives which manipulate buffers also manipulate the ready queue. All are finite queues represented as list structures.

```
Definition {1490}.
(AK-CLOCK-INTERRUPT-HANDLER AK)
   =
(AK-DISPATCHER
  (AK (AK-PSTATES AK)
      (AK-IBUFFERS AK)
      (AK-OBUFFERS AK)
      (AK-MBUFFERS AK)
      (ENQ (AK-TASKID AK) (DEQ (AK-READYQ AK)))
      (AK-STATUS AK)
      (AK-RWSTATE AK)
      (AK-CLOCK AK)
      (AK-IPORTS AK)
      (AK-OPORTS AK)))
Definition {1489}.
(AK-DISPATCHER AK)
   =
(AK (AK-PSTATES AK)
    (AK-IBUFFERS AK)
    (AK-OBUFFERS AK)
    (AK-MBUFFERS AK)
    (AK-READYQ AK)
    (AK-STATUS AK)
    (IF (QEMPTYP (AK-READYQ AK))
        (AK-WAIT-STATE)
        (AK-RUN-STATE))
    (IF (QEMPTYP (AK-READYQ AK))
        (AK-CLOCK AK)
        (AK-TIME-SLICE))
    (AK-IPORTS AK)
    (AK-OPORTS AK))
```

3.2.2 The Error Handler

A clock interrupt does not change the length of the ready queue or the status of a task. The error trap mechanism illustrates these situations. The error handler aborts the current task and prevents it from running again by removing its identifier from the head of the ready queue and updating its status to indicate an error condition. The status is updated by storing the 2-tuple **`(LIST (AK-ERROR-STATUS) 0)`** in the entry of **`AK-STATUS`** indexed by the current task identifier. An element of the status array is a 2-tuple *(status-flag taskid)*. A task's status is one of *ready*, *error*, *waiting-to-send*, *waiting-to-receive*, *waiting-to-input* or *waiting-to-output*. When a task is marked waiting to send or receive, the identifier of the task upon which it is waiting is recorded in the second element of the status tuple. For the other *status-flag* values a **`0`** is stored in the second element.

```
Definition {1491}.
(AK-ERROR-HANDLER AK)
   =
(AK-DISPATCHER
  (AK (AK-PSTATES AK)
      (AK-IBUFFERS AK)
      (AK-OBUFFERS AK)
      (AK-MBUFFERS AK)
      (DEQ (AK-READYQ AK))
      (PUTNTH (LIST (AK-ERROR-STATUS) 0)
              (AK-TASKID AK)
              (AK-STATUS AK))
      (AK-RWSTATE AK)
      (AK-CLOCK AK)
      (AK-IPORTS AK)
      (AK-OPORTS AK)))
```

3.2.3 The Supervisor Call Handler

The function **AK-SVC-HANDLER** interprets a request for one of a set of services provided by the kernel. These are exactly the communication primitives defined at the task layer: *send*, *receive*, *input* and *output*. **AK-SVC-HANDLER** itself is just a case split on the requested service. The functions **AK-SRCID**, **AK-DESTID** and **AK-MESSAGE** define conventions by which tasks pass arguments to the supervisor call handler.

The functions which define the services are given below. These services perform transitions on the buffers. In addition, they define operations on the kernel data structures which manage task activations.

```
Definition {1504}.
(AK-SVC-HANDLER AK)
   =
(IF (AK-SEND-INSTRUCTIONP AK)
    (AK-EXECUTE-SEND (AK-MESSAGE AK) (AK-TASKID AK) (AK-DESTID AK) AK)

(IF (AK-RECEIVE-INSTRUCTIONP AK)
    (AK-EXECUTE-RECEIVE (AK-SRCID AK) (AK-TASKID AK) AK)

(IF (AK-TYO-INSTRUCTIONP AK)
    (AK-EXECUTE-OUTPUT (AK-MESSAGE AK) (AK-TASKID AK) AK)

    (AK-EXECUTE-INPUT (AK-TASKID AK) AK))))
```

3.2.3.1. Send

The form **(AK-EXECUTE-SEND MSG SRCID DESTID AK)** gives an **AK** state which defines the *send* transition. If the buffer which implements communication from task **SRCID** to task **DESTID** is full, then the sending task is made to wait. Otherwise, the message

is delivered and the destination task is made ready if it had been waiting for a message from the sender. The function **AK-UPDATE-CONTROL** updates the control state of the sending task to step beyond the *send* request.

```
Definition {1494}.
(AK-EXECUTE-SEND MSG SRCID DESTID AK)
   =
(IF (QFULLP2 SRCID DESTID (AK-MBUFFERS AK) (TASK-MBUFFER-CAPACITY))
    (AK-BLOCK-SEND SRCID DESTID AK)
    (AK-EXECUTE-SEND-TO-BUFFER MSG SRCID DESTID AK))

Definition {1492}.
(AK-BLOCK-SEND SRCID DESTID AK)
   =
(AK-DISPATCHER
  (AK (AK-PSTATES AK)
      (AK-IBUFFERS AK)
      (AK-OBUFFERS AK)
      (AK-MBUFFERS AK)
      (DEQ (AK-READYQ AK))
      (PUTNTH (LIST (AK-SEND-STATUS) DESTID)
              SRCID
              (AK-STATUS AK))
      (AK-RWSTATE AK)
      (AK-CLOCK AK)
      (AK-IPORTS AK)
      (AK-OPORTS AK)))

Definition {1493}.
(AK-EXECUTE-SEND-TO-BUFFER MSG SRCID DESTID AK)
   =
(AK (AK-UPDATE-CONTROL SRCID (AK-PSTATES AK))
    (AK-IBUFFERS AK)
    (AK-OBUFFERS AK)
    (ENQ2 MSG SRCID DESTID (AK-MBUFFERS AK))
    (IF (AK-WAITING-TO-RECEIVEP SRCID DESTID AK)
        (ENQ DESTID (AK-READYQ AK))
        (AK-READYQ AK))
    (IF (AK-WAITING-TO-RECEIVEP SRCID DESTID AK)
        (PUTNTH (LIST (AK-READY-STATUS) 0)
                DESTID
                (AK-STATUS AK))
        (AK-STATUS AK))
    (AK-RWSTATE AK)
    (AK-CLOCK AK)
    (AK-IPORTS AK)
    (AK-OPORTS AK))
```

3.2.3.2. Receive

The form **(AK-EXECUTE-RECEIVE SRCID DESTID AK)** gives an **AK** state which defines the *receive* operation. If the buffer which implements communication from task **SRCID** to task **DESTID** is empty, then the receiving task is made to wait. Otherwise, the message is dequeued from the buffer and delivered to the receiving task. If the sender is waiting on a full buffer, it is made ready again. The function **AK-STORE-MESSAGE** defines the convention by which messages are delivered to the private state of a task.

```
Definition {1497}.
(AK-EXECUTE-RECEIVE SRCID DESTID AK)
   =
(IF (QEMPTYP2 SRCID DESTID (AK-MBUFFERS AK))
    (AK-BLOCK-RECEIVE SRCID DESTID AK)
    (AK-EXECUTE-RECEIVE-FROM-BUFFER SRCID DESTID AK))

Definition {1495}.
(AK-BLOCK-RECEIVE SRCID DESTID AK)
   =
(AK-DISPATCHER
  (AK (AK-PSTATES AK)
      (AK-IBUFFERS AK)
      (AK-OBUFFERS AK)
      (AK-MBUFFERS AK)
      (DEQ (AK-READYQ AK))
      (PUTNTH (LIST (AK-RECEIVE-STATUS) SRCID)
              DESTID
              (AK-STATUS AK))
      (AK-RWSTATE AK)
      (AK-CLOCK AK)
      (AK-IPORTS AK)
      (AK-OPORTS AK)))

Definition {1496}.
(AK-EXECUTE-RECEIVE-FROM-BUFFER SRCID DESTID AK)
   =
(AK (AK-UPDATE-CONTROL
      DESTID
      (AK-STORE-MESSAGE (QFIRST2 SRCID DESTID (AK-MBUFFERS AK))
                        DESTID
                        (AK-PSTATES AK)))
    (AK-IBUFFERS AK)
    (AK-OBUFFERS AK)
    (DEQ2 SRCID DESTID (AK-MBUFFERS AK))
    (IF (AK-WAITING-TO-SENDP SRCID DESTID AK)
        (ENQ SRCID (AK-READYQ AK))
        (AK-READYQ AK))
    (IF (AK-WAITING-TO-SENDP SRCID DESTID AK)
        (PUTNTH (LIST (AK-READY-STATUS) 0)
                SRCID
                (AK-STATUS AK))
        (AK-STATUS AK))
    (AK-RWSTATE AK)
    (AK-CLOCK AK)
    (AK-IPORTS AK)
    (AK-OPORTS AK))
```

3.2.3.3. Input

The input supervisor service handles a request by a task for a character from an input device. The abstract kernel buffers characters arriving from each input port and delivers them to the owning task on request. The function **AK-EXECUTE-INPUT** defines the input supervisor service. It accesses the input buffer indexed by the formal argument **ID**. If the buffer is empty, it blocks the requesting task. Otherwise, it removes the first character on the device input buffer and delivers it to the requesting task.

```
Definition {1503}.
(AK-EXECUTE-INPUT ID AK)
   =
(IF (QEMPTYP (GETNTH ID (AK-IBUFFERS AK)))
    (AK-BLOCK-INPUT ID AK)
    (AK-EXECUTE-INPUT-FROM-BUFFER ID AK))

Definition {1501}.
(AK-BLOCK-INPUT ID AK)
   =
(AK-DISPATCHER
  (AK (AK-PSTATES AK)
      (AK-IBUFFERS AK)
      (AK-OBUFFERS AK)
      (AK-MBUFFERS AK)
      (DEQ (AK-READYQ AK))
      (PUTNTH (LIST (AK-INPUT-STATUS) 0)
              ID
              (AK-STATUS AK))
      (AK-RWSTATE AK)
      (AK-CLOCK AK)
      (AK-IPORTS AK)
      (AK-OPORTS AK)))

Definition {1502}.
(AK-EXECUTE-INPUT-FROM-BUFFER ID AK)
   =
(AK (AK-UPDATE-CONTROL
      ID
      (AK-STORE-MESSAGE (QFIRST (GETNTH ID (AK-IBUFFERS AK)))
                        ID
                        (AK-PSTATES AK)))
    (PUTNTH (DEQ (GETNTH ID (AK-IBUFFERS AK)))
            ID
            (AK-IBUFFERS AK))
    (AK-OBUFFERS AK)
    (AK-MBUFFERS AK)
    (AK-READYQ AK)
    (AK-STATUS AK)
    (AK-RWSTATE AK)
    (AK-CLOCK AK)
    (AK-IPORTS AK)
    (AK-OPORTS AK))
```

3.2.3.4. Output

The output supervisor service handles a request by a task to send a character to an output device. The abstract kernel buffers characters waiting to be sent to a device, delivering one each time an output buffer is non-empty and its associated device is idle. The function **AK-EXECUTE-OUTPUT** defines the output supervisor service. It accesses the output buffer indexed by the formal argument **ID**. If the buffer is full, the requesting task is blocked. Otherwise, a character is enqueued on the buffer. If the associated device is idle, an output interrupt is triggered, causing the output interrupt handler to initiate an output to the device. I/O ports and I/O interrupts at the abstract kernel layer are defined to coincide with the implementation at the target machine layer.

```
Definition {1500}.
(AK-EXECUTE-OUTPUT CHAR ID AK)
   =
(IF (QFULLP (GETNTH ID (AK-OBUFFERS AK))
            (TASK-OBUFFER-CAPACITY))
    (AK-BLOCK-OUTPUT ID AK)
    (AK-EXECUTE-OUTPUT-TO-BUFFER CHAR ID AK))

Definition {1498}.
(AK-BLOCK-OUTPUT ID AK)
   =
(AK-DISPATCHER
  (AK (AK-PSTATES AK)
      (AK-IBUFFERS AK)
      (AK-OBUFFERS AK)
      (AK-MBUFFERS AK)
      (DEQ (AK-READYQ AK))
      (PUTNTH (LIST (AK-OUTPUT-STATUS) 0)
              ID
              (AK-STATUS AK))
      (AK-RWSTATE AK)
      (AK-CLOCK AK)
      (AK-IPORTS AK)
      (AK-OPORTS AK)))

Definition {1499}.
(AK-EXECUTE-OUTPUT-TO-BUFFER CHAR ID AK)
   =
(AK (AK-UPDATE-CONTROL ID (AK-PSTATES AK))
    (AK-IBUFFERS AK)
    (ENQ-ITH-BUFFER CHAR ID (AK-OBUFFERS AK))
    (AK-MBUFFERS AK)
    (AK-READYQ AK)
    (AK-STATUS AK)
    (AK-RWSTATE AK)
    (AK-CLOCK AK)
    (AK-IPORTS AK)
    (IF (AK-OPORT-IDLEP ID (AK-OPORTS AK))
        (AK-POST-OUTPUT-INTERRUPT ID (AK-OPORTS AK))
        (AK-OPORTS AK)))
```

3.2.4 The Input Interrupt Handler

An input interrupt is a non-deterministic event supplied by **AK**'s oracle. It signals the arrival of a character from an input device. The main functions of the input interrupt handler are: to enqueue the arriving input character on the designated buffer, to clear the input interrupt signal, and to make the owning task ready if it is waiting for input. The state in which the input interrupt handler leaves the kernel depends on whether the kernel is waiting. When waiting, the ready queue is empty. If the task which owns the interrupting input device is waiting on input that task is made ready and is dispatched, otherwise the kernel remains waiting. If the kernel is running, the current task is resumed without calling the dispatcher.

```
Definition {1509}.
(AK-INPUT-INTERRUPT-HANDLER ID AK)
   =
(IF (AK-WAITING AK)
    (AK-WAITING-INPUT-INTERRUPT-HANDLER ID AK)
    (AK-RUNNING-INPUT-INTERRUPT-HANDLER ID AK))

Definition {1507}.
(AK-WAITING-INPUT-INTERRUPT-HANDLER ID AK)
   =
(AK-DISPATCHER
  (AK (AK-PSTATES AK)
      (AK-UPDATE-IBUFFER ID AK)
      (AK-OBUFFERS AK)
      (AK-MBUFFERS AK)
      (IF (AK-WAITING-TO-INPUTP ID AK)
          (ENQ ID (AK-READYQ AK))
          (AK-READYQ AK))
      (IF (AK-WAITING-TO-INPUTP ID AK)
          (PUTNTH (LIST (AK-READY-STATUS) 0) ID (AK-STATUS AK))
          (AK-STATUS AK))
      (AK-RWSTATE AK)
      (AK-CLOCK AK)
      (AK-CLEAR-INPUT-INTERRUPT ID (AK-IPORTS AK))
      (AK-OPORTS AK)))

Definition {1508}.
(AK-RUNNING-INPUT-INTERRUPT-HANDLER ID AK)
   =
(AK (AK-PSTATES AK)
    (AK-UPDATE-IBUFFER ID AK)
    (AK-OBUFFERS AK)
    (AK-MBUFFERS AK)
    (IF (AK-WAITING-TO-INPUTP ID AK)
        (ENQ ID (AK-READYQ AK))
        (AK-READYQ AK))
    (IF (AK-WAITING-TO-INPUTP ID AK)
        (PUTNTH (LIST (AK-READY-STATUS) 0) ID (AK-STATUS AK))
        (AK-STATUS AK))
    (AK-RWSTATE AK)
    (AK-CLOCK AK)
    (AK-CLEAR-INPUT-INTERRUPT ID (AK-IPORTS AK))
    (AK-OPORTS AK))
```

The function **AK-UPDATE-IBUFFER** updates the input buffer. The I/O interface does not allow the kernel to make an input device wait. The condition of overflow is signaled by delivering to the buffer an *overflow character*, which is a message larger than the greatest possible character. This gives the owning task a method of detecting overflow. If an input buffer is full, **AK-UPDATE-IBUFFER** replaces the last character on the queue with an overflow character. If the buffer is not full but the input port indicates an overflow, an overflow character is enqueued on the input buffer. Otherwise, no overflow error has occurred either at the buffer or port, and the character is enqueued.

3.2.5 The Output Interrupt Handler

An output interrupt signals that an output has been completed and an output device is idle. Our definition of the abstract kernel is not comprehensive enough to specify the precise relationship between a command to start output to a device and the corresponding output interrupt signaling completion of the output. Output interrupts can be treated only as non-deterministic events supplied by **AK**'s oracle.

An output interrupt transition is defined as follows. In all cases, the output interrupt is cleared. If the corresponding output buffer is non-empty, then a new output is started. If the owning task had been waiting on a full output buffer, it is made ready again. The conditions of full buffer and empty buffer are mutually exclusive, so a task cannot be waiting when a buffer is empty. Like the input interrupt handler, the state in which the output interrupt handler leaves the kernel also depends on whether or not the kernel is waiting. When waiting, the ready queue is empty. If the task which owns the interrupting output device is waiting on output that task is made ready and is dispatched, otherwise the kernel remains waiting. If the kernel is running, the current task is resumed without calling the dispatcher.

```
Definition {1512}.
(AK-OUTPUT-INTERRUPT-HANDLER ID AK)
   =
(IF (AK-WAITING AK)
    (AK-WAITING-OUTPUT-INTERRUPT-HANDLER ID AK)
    (AK-RUNNING-OUTPUT-INTERRUPT-HANDLER ID AK))
```

```
Definition {1510}.
(AK-WAITING-OUTPUT-INTERRUPT-HANDLER ID AK)
   =
(AK-DISPATCHER
  (AK (AK-PSTATES AK)
      (AK-IBUFFERS AK)
      (IF (QEMPTYP (GETNTH ID (AK-OBUFFERS AK)))
          (AK-OBUFFERS AK)
          (DEQ-ITH-BUFFER ID (AK-OBUFFERS AK)))
      (AK-MBUFFERS AK)
      (IF (AK-WAITING-TO-OUTPUTP ID AK)
          (ENQ ID (AK-READYQ AK))
          (AK-READYQ AK))
      (IF (AK-WAITING-TO-OUTPUTP ID AK)
          (PUTNTH '(0 0) ID (AK-STATUS AK))
          (AK-STATUS AK))
      (AK-RWSTATE AK)
      (AK-CLOCK AK)
      (AK-IPORTS AK)
      (IF (QEMPTYP (GETNTH ID (AK-OBUFFERS AK)))
          (AK-CLEAR-OUTPUT-INTERRUPT ID (AK-OPORTS AK))
          (AK-START-OUTPUT (QFIRST (GETNTH ID (AK-OBUFFERS AK)))
                           ID
                           (AK-OPORTS AK)))))

Definition {1511}.
(AK-RUNNING-OUTPUT-INTERRUPT-HANDLER ID AK)
   =
(AK (AK-PSTATES AK)
    (AK-IBUFFERS AK)
    (IF (QEMPTYP (GETNTH ID (AK-OBUFFERS AK)))
        (AK-OBUFFERS AK)
        (DEQ-ITH-BUFFER ID (AK-OBUFFERS AK)))
    (AK-MBUFFERS AK)
    (IF (AK-WAITING-TO-OUTPUTP ID AK)
        (ENQ ID (AK-READYQ AK))
        (AK-READYQ AK))
    (IF (AK-WAITING-TO-OUTPUTP ID AK)
        (PUTNTH '(0 0) ID (AK-STATUS AK))
        (AK-STATUS AK))
    (AK-RWSTATE AK)
    (AK-CLOCK AK)
    (AK-IPORTS AK)
    (IF (QEMPTYP (GETNTH ID (AK-OBUFFERS AK)))
        (AK-CLEAR-OUTPUT-INTERRUPT ID (AK-OPORTS AK))
        (AK-START-OUTPUT (QFIRST (GETNTH ID (AK-OBUFFERS AK)))
                         ID
                         (AK-OPORTS AK))))
```

This concludes our excursion through the definition of the abstract kernel. The remaining details of the kernel's definition occur in the proof script. Like the task layer, **AK** relies on the target machine's definition of the fetch-execute step on the private state of a task. It also uses the target machine's implementation of communication with I/O devices. **AK** is abstract in the following ways.

- The private state spaces of tasks are transparently isolated. This provides an important constraint on the implementation.

- The data structures used to manage tasks are represented as high-level list structures.
- The transitions on the kernel state are specified functionally. All kernel operations take place in a single abstract step.

Chapter 4

The Implementation of KIT

In this chapter we define the target machine upon which we implement KIT. We then present the kernel source code. We include the code in the text not because we find it particularly readable, but because the existence of this verified low-level code is one of the most important characteristics of this work.

4.1 The Target Machine

We arrive at the bottom rung of the ladder in Figure 2-2 to discuss the target machine `TM`. The target machine is a simple von Neumann computer. It is not based on any existing physical machine because we are not interested in the task of formalizing an existing machine. We intend for `TM` to be straightforward.

`TM` has simple architectural support for multi-programming. This support consists of a base/limit register pair mechanism for memory protection, and a supervisor/user mode flag for protecting privileged operations. `TM` is a 16-bit machine. Main memory consists of 2^{16} 16-bit words. The processor state contains 8 general purpose registers, one of which is the program counter and another a stack pointer. There are four flag fields: a 2-bit condition code, a 6-bit error code, a supervisor call flag, and a 7-bit supervisor call identifier. Processor registers which are accessible only in the supervisor mode are the base/limit register pair, a supervisor address limit register, the supervisor/user mode flag, a running/wait state flag and the program clock. `TM` is capable of asynchronous character I/O. It communicates with 16 input devices and 16 output devices by an array of input ports and an array of output ports. Table PMS gives a summary of the `TM` architecture in PMS notation [PMS].

```
Memory state

  Mp[0:65535]<0:15>   main memory of 2^16 16-bit words

Pc state

  R[0:7]<0:15>        8 general purpose registers;
                      R[0] is the PC; R[1] is the SP

  CC<0:1>             2-bit condition code
  ERROR<0:5>          6-bit error code
  SVCFLAG             1-bit svc call flag
  SVCID<0:6>          7-bit svc identifier

  BASE<0:15>          16-bit address base register
  LIMIT<0:15>         16-bit address limit register
  SLIMIT<0:15>        16-bit address defining the upper limit
                      of the supervisor based at address 0 in
                      memory

  SVMODE              supervisor/user mode flag
  RWSTATE             running/wait state flag
  CLOCK<0:15>         program clock used for time slicing

I/O interface

  IPORTS[0:15](<0:1>;<0:1>;<0:7>) an array of 16 input ports;
                                  each port is a 3-tuple
                                  (interrupt-flag, error-flag, character-buffer)

  OPORTS[0:15](<0:1>;<0:1>;<0:7>) an array of 16 output ports;
                                  each port is a 3-tuple
                                  (interrupt-flag, busy-flag, character-buffer)
```

Table 4-1: PMS Description of TM

The structure of the target machine is described in the Boyer-Moore logic by the shell **TM**. The fields defined by the shell correspond to the fields described in Table PMS.

```
Shell Definition {668}.
Add the shell TM with recognizer TM-SHELLP,
defining the record structure
<TM-MEMORY, TM-REGS, TM-CC, TM-ERROR, TM-SVCFLAG, TM-SVCID,
 TM-BASE, TM-LIMIT, TM-SLIMIT, TM-SVMODE, TM-RWSTATE, TM-CLOCK,
 TM-IPORTS, TM-OPORTS>.
```

The predicate **GOOD-TM** defines the target machine state space. Each **TM** component is represented as a natural number, a list of natural numbers, or, in the case of I/O ports, a tuple of natural numbers. The maximum sizes of the components are defined by constant functions, some of which are given below. This is a slightly more abstract

representation of a machine than one which uses sequences of bits (*bit vectors*). We justify this level of abstraction by observing that there is a 1-1 mapping between bit vectors of a given size and the set of natural numbers from 0 to the maximum number representable by the bit vector. Therefore the natural number representation for a machine is isomorphic to a bit vector representation. In addition, if we had chosen to go all the way down to a bit vector representation we would have been obliged to verify **TM**'s ALU, a problem treated by Hunt [Hunt85] and beyond the scope of this work.

```
Definition {907}.
(GOOD-TM TM)
   =
(AND (TM-SHELLP TM)
     (PLISTP (TM-MEMORY TM))
     (EQUAL (LENGTH (TM-MEMORY TM)) (TM-MEMLENGTH))
     (FINITE-NUMBER-LISTP (TM-MEMORY TM)
                          (TM-WORDLUB))
     (PLISTP (TM-REGS TM))
     (EQUAL (LENGTH (TM-REGS TM)) (TM-REGLENGTH))
     (FINITE-NUMBER-LISTP (TM-REGS TM) (TM-WORDLUB))
     (FINITE-NUMBERP (TM-CC TM) (TM-CCLUB))
     (FINITE-NUMBERP (TM-ERROR TM) (TM-ERRORLUB))
     (FINITE-NUMBERP (TM-SVCFLAG TM) (TM-SVCFLAGLUB))
     (FINITE-NUMBERP (TM-SVCID TM) (TM-SVCIDLUB))
     (FINITE-NUMBERP (TM-BASE TM) (TM-WORDLUB))
     (FINITE-NUMBERP (TM-LIMIT TM) (TM-WORDLUB))
     (FINITE-NUMBERP (TM-SLIMIT TM) (TM-WORDLUB))
     (FINITE-NUMBERP (TM-SVMODE TM) 2)
     (FINITE-NUMBERP (TM-RWSTATE TM) 2)
     (FINITE-NUMBERP (TM-CLOCK TM) (TM-WORDLUB))
     (PLISTP (TM-IPORTS TM))
     (EQUAL (LENGTH (TM-IPORTS TM))(TM-PORT-LENGTH))
     (GOOD-TM-IPORT-ARRAY (TM-IPORTS TM))
     (PLISTP (TM-OPORTS TM))
     (EQUAL (LENGTH (TM-OPORTS TM)) (TM-PORT-LENGTH))
     (GOOD-TM-OPORT-ARRAY (TM-OPORTS TM)))

Definition {613}.
(TM-WORDSIZE) = 16

Definition {614}.
(TM-WORDLUB) = (EXP 2 (TM-WORDSIZE))

Definition {682}.
(TM-PORT-LENGTH) = 16
```

The structure of input and output ports is formalized using shells. An input port is a 3-tuple containing an interrupt flag, an error flag which is used to indicate overflow on the input port, and a character buffer. An output port is a 3-tuple containing an interrupt flag, a busy flag, and a character buffer. The functions **GOOD-TM-IPORT-ARRAY** and **GOOD-TM-OPORT-ARRAY** recognize fixed-length arrays of I/O ports with bounded components.

```
Shell Definition {748}.
Add the shell TM-IPORT with recognizer TM-IPORTP,
defining the record structure
<TM-IINTERRUPT-FLAG, TM-IERROR-FLAG, TM-ICHAR>.

Shell Definition {749}.
Add the shell TM-OPORT with recognizer TM-OPORTP,
defining the record structure
<TM-OINTERRUPT-FLAG, TM-OBUSY-FLAG, TM-OCHAR>.
```

The function **TM-PROCESSOR** is the interpreter function which defines the transitions on a **TM** state. The formal argument **TM** represents a machine state, and the formal argument **ORACLE** represents an oracle identical to an abstract kernel oracle. That is, an oracle is a list some of whose elements are I/O interrupts. An input interrupt is a 2-tuple which gives an input character and a device id, accessed by the functions **TM-IDATUM** and **TM-IDEVID**, respectively. An output interrupt merely contains a device id, accessed by the function **TM-ODEVID**.

```
Definition {883}.
(TM-PROCESSOR TM ORACLE)
   =
(IF (LISTP ORACLE)
    (TM-PROCESSOR (TM-STEP (TM-POST-INTERRUPT (CAR ORACLE) TM))
                  (CDR ORACLE))
    TM)
```

TM-POST-INTERRUPT incorporates interrupts into the state of the machine so that they can be sensed. An input interrupt for device *i* is posted by changing the value of the *i*th input port as follows: the interrupt flag is raised, the error flag gets the previous value of the interrupt flag to signal an overflow condition, the input character is written to the character buffer. An output interrupt for device *i* is posted by changing the value of the *i*th output port as follows: the interrupt flag is raised, the busy flag is cleared, the character buffer is cleared (although this action is superfluous). When the current oracle element is not an I/O interrupt, **TM-POST-INTERRUPT** makes no change to the state of the machine.

```
Definition {881}.
(TM-POST-INTERRUPT EVENT TM)
   =
(IF (TM-DEVICE-INPUT-EVENTP EVENT)
    (TM-SET-IPORTS (TM-POST-INPUT-INTERRUPT (REMAINDER (TM-IDATUM EVENT)
                                                       (TM-CHARLUB))
                                           (REMAINDER (TM-IDEVID EVENT)
                                                      (TM-PORT-LENGTH))
                                           (TM-IPORTS TM))
                   TM)
    (IF (TM-DEVICE-OUTPUT-EVENTP EVENT)
        (TM-SET-OPORTS (TM-POST-OUTPUT-INTERRUPT (REMAINDER (TM-ODEVID E
                                                            (TM-PORT-LEN
                                                 (TM-OPORTS TM))
                       TM)
        TM))

Definition {752}.
(TM-POST-INPUT-INTERRUPT CHAR ID PORTS)
   =
(PUTNTH (TM-IPORT 1
                  (TM-IINTERRUPT-FLAG (GETNTH ID PORTS))
                  CHAR)
        ID
        PORTS)

Definition {755}.
(TM-POST-OUTPUT-INTERRUPT ID PORTS)
   =
(PUTNTH (TM-OPORT 1 0 0) ID PORTS)
```

The function **TM-STEP** defines the single step function for the **TM** interpreter. It gives the interrupt structure of the target machine. Each of the interrupt branches of **TM-STEP** (an input interrupt, an output interrupt, an error trap, a clock interrupt and a supervisor call interrupt) does a *PSW swap*, which partially saves the state of the CPU in a fixed location of memory and loads a new program counter giving the address of an operating system interrupt handling routine. When no I/O interrupt occurs and TM is in the wait state, **TM-STEP** returns the current machine state unchanged. The function **TM-FETCH-EXECUTE** defines the instruction fetch-execute cycle of the target machine.

```
Definition {882}.
(TM-STEP TM)
   =
(IF (TM-INPUT-INTERRUPTP TM)
    (TM-EXECUTE-INPUT-INTERRUPT TM)

(IF (TM-OUTPUT-INTERRUPTP TM)
    (TM-EXECUTE-OUTPUT-INTERRUPT TM)

(IF (TM-WAITING TM)
    TM

(IF (TM-ERRORP TM)
    (TM-EXECUTE-ERROR-INTERRUPT TM)

(IF (TM-CLOCK-INTERRUPTP TM)
    (TM-EXECUTE-CLOCK-INTERRUPT TM)

(IF (TM-SVC-INTERRUPTP TM)
    (TM-EXECUTE-SVC-INTERRUPT TM)

    (TM-FETCH-EXECUTE TM)))))))
```

We wish to examine interrupts and the fetch-execute cycle more closely. Before doing so, we examine some of the primitive functions in the definition of **TM** which update the **TM** state. In particular, we examine memory and register access. First, for every field in the **TM** structure we have defined a function which updates that field and no other. For instance, the function **TM-SET-CC** returns a **TM** state with an updated condition code.

```
Definition {687}.
(TM-SET-CC CC TM)
   =
(TM (TM-MEMORY TM)
    (TM-REGS TM)
    CC
    (TM-ERROR TM)
    (TM-SVCFLAG TM)
    (TM-SVCID TM)
    (TM-BASE TM)
    (TM-LIMIT TM)
    (TM-SLIMIT TM)
    (TM-SVMODE TM)
    (TM-RWSTATE TM)
    (TM-CLOCK TM)
    (TM-IPORTS TM)
    (TM-OPORTS TM))
```

The interface to memory and register access is defined by the functions **TM-FETCH** and **TM-STORE**. An address argument to these functions is a 2-tuple constructed by the function **REAL-ADDR**. The **REAL-ADDR-NUM** field is a number used as a datum or an

address. The **REAL-ADDR-SOURCE** field indicates how the number is used: as a datum, as a register address, or as a memory address. **TM-FETCH** and **TM-STORE** follow the convention that a **REAL-ADDR-SOURCE** value of **0** indicates a datum, a value of **1** indicates a register address, and otherwise a memory address. Given a **REAL-ADDR**, **TM-FETCH** returns either the datum portion of the address, or the contents of a register, or the contents of a memory word. **TM-STORE** makes no state change when given a **REAL-ADDR** with source **0**, and otherwise updates a location in either the registers or memory. Notice that when the machine is in user mode, a memory address is treated as a displacement from the current base register.

```
Definition {773}.
(TM-FETCH ADDR TM)
   =
(IF (ZEROP (REAL-ADDR-SOURCE ADDR))
    (REAL-ADDR-NUM ADDR)

(IF (EQUAL (REAL-ADDR-SOURCE ADDR) 1)
    (TM-FETCH-FROM-REGMEM (REAL-ADDR-NUM ADDR) TM)

    (TM-FETCH-FROM-MEMORY (REAL-ADDR-NUM ADDR) TM)))

Definition {774}.
(TM-STORE VALUE ADDR TM)
   =
(IF (ZEROP (REAL-ADDR-SOURCE ADDR))
    TM

(IF (EQUAL (REAL-ADDR-SOURCE ADDR) 1)
    (TM-STORE-IN-REGMEM VALUE (REAL-ADDR-NUM ADDR) TM)

    (TM-STORE-IN-MEMORY VALUE (REAL-ADDR-NUM ADDR) TM)))

Definition {757}.
(REAL-ADDR SOURCE NUM) = (LIST SOURCE NUM)

Definition {758}.
(REAL-ADDR-SOURCE REAL-ADDR) = (CAR REAL-ADDR)

Definition {759}.
(REAL-ADDR-NUM REAL-ADDR) = (CADR REAL-ADDR)

Definition {769}.
(TM-FETCH-FROM-MEMORY ADDR TM)
   =
(IF (TM-IN-SUPERVISOR-MODE TM)
    (GETNTH ADDR (TM-MEMORY TM))
    (GETNTH (PLUS (TM-BASE TM) ADDR) (TM-MEMORY TM)))
```

```
Definition {770}.
(TM-STORE-IN-MEMORY VALUE ADDR TM)
   =
(IF (TM-IN-SUPERVISOR-MODE TM)
    (TM-SET-MEMORY (PUTNTH VALUE ADDR (TM-MEMORY TM)) TM)
    (TM-SET-MEMORY (PUTNTH VALUE
                           (PLUS (TM-BASE TM) ADDR)
                           (TM-MEMORY TM))
                   TM))
Definition {771}.
(TM-FETCH-FROM-REGMEM ADDR TM)
   =
(GETNTH ADDR (TM-REGS TM))
Definition {772}.
(TM-STORE-IN-REGMEM VALUE ADDR TM)
   =
(TM-SET-REGS (PUTNTH VALUE ADDR (TM-REGS TM)) TM)
```

Now we return to the subject of interrupts. Table CLOCK-INTERRUPT describes what happens on a clock interrupt: the current program counter, stack pointer and flags fields are stored in memory locations [0:2]. A new program counter is loaded from a fixed location in memory giving the address of the clock interrupt handler, the stack pointer is loaded with the supervisor limit address (a stack occupies the high address end of a memory segment), and the machine is put in supervisor mode.

```
mem[0:2]   <-  [pc,sp,flags]
pc         <-  mem[3]
sp         <-  slimit - 1
svmode     <-  supervisor-mode
```

Table 4-2: The TM Clock Interrupt

We explore the formal definition of the clock interrupt given by **TM-EXECUTE-CLOCK-INTERRUPT**. The machine is put in supervisor mode (the **TM-SET-SVMODE** expression), the program counter, stack pointer, and flags are saved in location 0 through 2 of memory (the call to **TM-STORE-OLD-PSW-ON-INTERRUPT**), the program counter is loaded with a new value (the call to **TM-FETCH-NEW-PC-ON-INTERRUPT**), and the stack pointer is set to one less than the supervisor limit register (the **TM-SET-SP** expression). In **TM-STORE-OLD-PSW-ON-INTERRUPT**, the function **TM-INCRN-ADDRESS** increments an address a given number of times. This is how we arrange to store CPU state in three successive memory locations. **TM-PACK-PSW** packs the flags fields into a single number. All of the other interrupt transitions referenced in **TM-STEP** are defined in a similar fashion.

```
Definition {862}.
(TM-EXECUTE-CLOCK-INTERRUPT TM)
   =
(TM-SET-SP (TM-DECR (TM-SLIMIT TM))
 (TM-FETCH-NEW-PC-ON-INTERRUPT (TM-CLOCK-NEW-PC-ADDR)
  (TM-STORE-OLD-PSW-ON-INTERRUPT (TM-REGISTER-SAVE-AREA-ADDR)
   (TM-SET-SVMODE (TM-SUPERVISOR-MODE)
                  TM))))

Definition {859}.
(TM-FETCH-NEW-PC-ON-INTERRUPT ADDR TM)
   =
(TM-SET-PC (TM-FETCH-FROM-MEMORY ADDR TM) TM)

Definition {860}.
(TM-STORE-OLD-PSW-ON-INTERRUPT ADDR TM)
   =
(TM-STORE (TM-PC TM)
          (REAL-ADDR 2 ADDR)
          (TM-STORE (TM-SP TM)
                    (TM-INCRN-ADDRESS 1 (REAL-ADDR 2 ADDR))
                    (TM-STORE (TM-PACK-PSW (TM-CC TM)
                                           (TM-ERROR TM)
                                           (TM-SVCFLAG TM)
                                           (TM-SVCID TM))
                              (TM-INCRN-ADDRESS 2 (REAL-ADDR 2 ADDR))
                              TM)))

Definition {721}.
(TM-REGISTER-SAVE-AREA-ADDR) = 0
```

We have seen the function **TM-FETCH-EXECUTE** referenced at the task and abstract kernel layers. It defines **TM**'s fetch-execute cycle. The function **TM-GOOD-PC-ADDRESS** determines if the address contained in the program counter causes a protection error as defined by the current contents of the limit register. If so, the error flag is set. Otherwise, the current instruction is fetched and executed. In addition, the program clock is decremented. The function **TM-EXECUTE** fetches the current instruction's arguments and computes absolute addresses based on the indicated address mode. **TM** has four address modes: immediate, memory direct, register, and register indirect. Memory addresses must be less than the current value of the limit register, otherwise causing a protection error. When running in user mode a memory address supplied by an instruction is treated as a displacement from the current base register.

```
Definition {858}.
(TM-FETCH-EXECUTE TM)
   =
(IF (TM-GOOD-PC-ADDRESS TM)
    (TM-EXECUTE (TM-FETCH-OPCODE TM) (TM-DECREMENT-CLOCK TM))
    (TM-SET-ERROR (TM-PC-ADDRESS-ERROR) (TM-DECREMENT-CLOCK TM)))
```

Table INSTRUCTION-SET documents **TM**'s small instruction set. The purpose of the table is to suggest the extent of the instruction set. We have defined only those instructions required to program the operating system. Other instructions can be added with the cost of proving that each one satisfies the **GOOD-TM** invariant. **TM** has instructions of zero, one and two arguments. The parameters which occur in Table INSTRUCTION-SET should be interpreted as real addresses: one of memory address, register address or immediate operand. In the case of binary operations, a result is stored at the location indicated by the first argument. The condition code is a 2-bit value which indicates two ALU conditions: zero/non-zero and carry/no-carry.

```
Non-Privileged Operations

ADD a b       add, set the condition code
BR a          set the pc unconditionally
BRZ a         set the pc if cc = <zero,non-overflow>
BRNZ a        set the pc if cc #≠# <zero,non-overflow>
CALL a        save the pc on the stack, load a new pc
COMPARE a b   set the condition code based on numerically
              comparing a and b
DECR a        decrement, set the condition code
DECR-MOD a b  decrement a modulo b, set the condition code
INCR a        increment, set the condition code
INCR-MOD a b  increment a modulo b, set the condition code
MOD a b       a mod b, set the condition code
MOVE a b      move b to location indicated by a
MULT a b      multiply, set the condition code
RETURN        set the pc to the top element of the stack
SVC addr      raise the svcflag, set the svcid

Privileged Operations

LBASE a       load the base register
LLIMIT a      load the limit register
LPSW a        load the pc, sp and flags; put the machine in user mode
POST a        raise the output interrupt flag in the output
              port given by the argument
RUN           put the machine in the run state
TIME a        set the clock
STOUT a b     start output on the device indicated by a;
              the output character is given by b
SVCR a        load the pc, sp and flags; put the machine
              in user mode; clear the svcflag
TESTI a       test the indicated input port for an overflow error
TESTO a       test the indicated output port for busy
WAIT          put the machine in the wait state
```

Table 4-3: TM's Instruction Set

We give the formal definition of the addition operation, **TM-EXECUTE-ADD**. It

takes three arguments, two real addresses indicating the addition operands, and the current state of the machine. **TM-EXECUTE-ADD** returns a new machine state by storing the the result of the addition at the location indicated by the first address, and updating the condition code to reflect two conditions: whether the result is zero, and whether the result has a carry out.

```
Definition {778}.
(TM-EXECUTE-ADD ADDR1 ADDR2 TM)
   =
(TM-STORE (ALU-VALUE (TM-ALU-PLUS (TM-FETCH ADDR1 TM)
                                  (TM-FETCH ADDR2 TM)))
          ADDR1
          (TM-SET-CC (TM-CC-VALUE (TM-ALU-PLUS (TM-FETCH ADDR1 TM)
                                               (TM-FETCH ADDR2 TM)))
                     TM))

Definition {707}.
(TM-CC-VALUE ALU-RESULT)
   =
(IF (ZEROP (ALU-VALUE ALU-RESULT))
    (IF (FALSEP (ALU-CARRY ALU-RESULT))
        (TM-ZERO-NO-CARRY-CONDITION)
        (TM-ZERO-CARRY-CONDITION))
    (IF (FALSEP (ALU-CARRY ALU-RESULT))
        (TM-NON-ZERO-NO-CARRY-CONDITION)
        (TM-NON-ZERO-CARRY-CONDITION)))
```

TM's ALU performs the following operations: *plus*, *difference*, *times*, *remainder*, *increment*, *decrement*, *increment-mod* and *decrement-mod*. *Increment-mod* takes two arguments and increments its first argument modulo its second argument. *Decrement-mod* decrements its first argument modulo its second argument. Besides returning an integer value, each ALU operation also sets a carry bit. Remainder is a powerful operation. The kernel in fact uses this operation only to take the remainder of a number by some power of two. Therefore the remainder operation in the ALU could be replaced by a simpler mask operation to satisfy the needs of the kernel.

This completes our summary of **TM**. It is a very simple von Neumann machine. It provides some support for the implementation of tasks, but cannot accomplish this on its own. The operating system kernel which must be written for **TM** has the significant job of spanning the gap to the abstract kernel.

4.2 The Code

In this section we present the source code of KIT. We present it as a listing in an assembler language written for **TM**, annotated with comments. A quoted list containing each line (minus comments) of the source code is equal to the function **OS-SOURCE**. This function, which appears in the script, is a constant function defining a list containing the assembler language source. The source code contains routines which correspond to the interrupt handlers specified at the abstract kernel level.

The kernel resides in a segment of memory beginning at location 0. Remaining memory segments are occupied by tasks. Figure LAYOUT-OF-KERNEL describes the memory layout of the kernel segment. It identifies the data structures required by the kernel.

- Register Save Area. This is a 3-word segment built into the definition of **TM** which is used to partially save the CPU state on an interrupt.
- Interrupt Vector. These addresses, also built into **TM**'s definition, contain the addresses of the interrupt handlers.
- Locals. A set of local variables used by the operating system.
- Task Table. This is a kernel data structure which contains the CPU state of each task.
- Segment Table. The table contains a base/limit register pair for each task, defining the location and length of each task's memory segment.
- Ready Queue. An implementation of the ready queue.
- Status Table. An implementation of the task status table.
- Ibuffer, Obuffers, Mbuffers. Implementations of the buffer tables.
- Code. The kernel machine code.
- Stack. The kernel's stack.

The assembler is a simple one written in the Boyer-Moore logic. It plays no part in the proof since we verify the output of the assembler, which is a list of numbers that **TM** interprets. The grammar accepted by the assembler is given in Table GRAMMAR. The primitives of the grammar are **<SYMBOL>** and **<NATNUM>**. **<NATNUM>** is understood to be a number bounded by **TM**'s wordsize. The grammar defines six forms. A

Figure 4-1: Layout of Kernel

<SYMBOL> makes an entry in the symbol table, associating a symbol with a displacement from the start of the source code. A **<DCL>** makes an entry in the symbol table to associate a symbol with a user supplied number. The **<DC>** form initializes a contiguous sequence of memory words and is used for declaring data storage. The remaining forms define the syntax of nullary, unary and binary operations. An **<ARG>** is a list containing an address mode, a value and an optional displacement. In the syntax for **<ARG>**, **<SYMBOL>** is an abbreviation for **(0 <SYMBOL> 0)** and **(<MODE> <VALUE>)** is an abbreviation for **(<MODE> <VALUE> 0)**.

This description of the grammar, plus the informal documentation of the

```
<GRAM>    ::= <FORM>*

<FORM>    ::= <SYMBOL> | <DCL> | <DC> | <0ARY-OP> |
              <1ARY-OP> | <2ARY-OP>

<DCL>     ::= (DCL <SYMBOL> <NATNUM>)

<DC>      ::= (DC <NATNUM> <VALUE>)

<0ARY-OP> ::= (<SYMBOL>)

<1ARY-OP> ::= (<SYMBOL> <ARG>)

<2ARY-OP> ::= (<SYMBOL> <ARG> <ARG>)

<ARG>     ::= <SYMBOL> | (<MODE> <VALUE>) |
              (<MODE> <VALUE> <DISP>)

<MODE>    ::= <NATNUM>[0..3]

<DISP>    ::= <NATNUM>[0..7]

<VALUE>   ::= <NATNUM> | <SYMBOL>
```

Table 4-4: Grammar for TM Assembler

instruction set in Table 4-3 should make it possible to read the assembler language source. Of course, all questions about details must be answered by consulting the definition of **TM** in the script. The assembler packs operations into one, two or three machine words. The format of machine instructions is not important. To be able to read the source code, the following facts should be understood about **TM**'s interpretation of instructions.

- The address modes are as follows: 0 - immediate operand, 1- register, 2 - memory, 3 - register indirect.
- Data movement in a binary operation is from right to left. For the instruction **(ADD A B)**, the sum of **A** and **B** is placed in the location indicated by **A** unless **A** is an immediate operand, in which case a result is not stored.
- Register 0 is the program counter, and register 1 is the stack pointer.

The source listing contains three sections. First is a series of **DCL** forms, defining symbols for the assembler. Next is a series of **DC** forms defining the data areas. The remainder contains programs.

We provide a guide to one part of the source, the clock interrupt handler, exhibiting small portions of the listing. Consult the definition of **AK-CLOCK-INTERRUPT**

handler in Section 3.2.1 for the specification. The clock interrupt handler begins at the address with label **CLOCK-INTERRUPT-HANDLER**. Control passes to the clock interrupt handler after a clock interrupt. The routine **SAVE-STATE** is called to save the state of the current task in the task table. Upon return, the clock interrupt handler loads the address of the ready queue into register 3, and then calls **QFIRST**, which places the first element of a queue into register 2. **DEQUEUE** is called to remove the first element from the ready queue. And then **ENQUEUE** is called to place what was the first queue element at the end of the queue. Finally, control branches to the dispatcher to resume the next task.

```
CLOCK-INTERRUPT-HANDLER
(call save-state)           ;; First, save the state of the current task.
trace-label1
(move (1 r3) readyq)        ;; R3 points to readyq
(call qfirst)               ;; Put current taskid in R2
(call dequeue)              ;; DEQUEUE the current task from the READYQ
(call enqueue)              ;; ENQUEUE the current task
trace-label2
(br dispatcher)             ;; Resume next task
```

The function **AK-DISPATCHER** in Section 3.2.1 specifies the dispatching operation. The dispatcher checks for an empty ready queue. If empty, the machine is put in the wait state. Otherwise, **QFIRST** is called to obtain the taskid which as the first element of the ready queue. **RESTORE-STATE** is called to initialize the CPU with most of this task's CPU state - all but the program counter, stack pointer and flags. Upon return, the program clock is reset, and an **LPSW** instruction is done to complete the context switch.

```
DISPATCHER ;; Allocate CPU to first task on readyq.
(move (1 r3) readyq)    ;; Point R3 to readyq
(call qemptyp)          ;; Readyq empty?
dispatcher-trace-label1
(brz  readyq-empty)
(call qfirst)           ;; Put next taskid in R2
(call restore-state)    ;; resume next task
dispatcher-trace-label2
(time (2 time-slice 0)) ;; set clock
(lpsw (2 reg-save-area))

READYQ-EMPTY
(wait)
```

```
;; ------------- Beginning of KIT source --------------
;; ---------- Assembler symbolic declarations ----------
(dcl r0 0)
(dcl r1 1)
(dcl r2 2)
(dcl r3 3)
(dcl r4 4)
(dcl r5 5)
(dcl r6 6)
(dcl r7 7)

;; format of interrupt save-area
(dcl interrupt-pc-field 0)
(dcl interrupt-sp-field 1)
(dcl interrupt-flag-field 2)
(dcl svcid-addr 8)
(dcl input-devid-addr 8)
(dcl input-char-addr 9)
(dcl output-devid-addr 9)
(dcl charlub 256)

;; svcids
(dcl send-svcid 0)
(dcl receive-svcid 1)
(dcl tyo-svcid 2)
(dcl tyi-svcid 3)

;; format of a task table entry
(dcl task-table-length 144)
(dcl task-table-entry-length 9)
(dcl pc-field 0)
(dcl sp-field 1)
(dcl r2-field 2)
(dcl r3-field 3)
(dcl r4-field 4)
(dcl r5-field 5)
(dcl r6-field 6)
(dcl r7-field 7)
(dcl flag-field 8) ;; displacement after bumping base register

;; format of a queue entry: [headaddr tailaddr currlength maxlength qarry]
;; where qarry is reserved for length maxlength
(dcl readyq-length 20)
(dcl qhead-field 0)
(dcl qtail-field 1)
(dcl qcurrlength-field 2)
(dcl qmaxlength-field 3)
(dcl qarray-field 4)

;; format of segment table
(dcl segment-table-length 32)
(dcl base-field 0)
(dcl limit-field 1)

;; format of status table
(dcl status-entry-length 2)
(dcl status-flag-field 0)
(dcl status-taskid-field 1)
(dcl ready-status 0)
(dcl error-status 1)
(dcl send-status 2)
```

```
(dcl receive-status 3)
(dcl output-status 4)
(dcl input-status 5)

;; Buffer lengths
(dcl input-buffer-length 8)
(dcl output-buffer-length 8)
(dcl message-buffer-length 8)

;; Values for access 2D array of message buffers
;; The address of MBUFFER[sourceid,destid] is
;; MBUFFERS + (sourceid * SOURCE-MULTIPLIER) + (destid * DEST-MULTIPLIER)
(dcl source-multiplier 128)
(dcl dest-multiplier 8)
(dcl taskidlub 16)

;; ---------- Data areas in operating system ----------
reg-save-area  (dc 3 0)  ;; [pc sp flags]
clock-new-pc   (dc 1 clock-interrupt-handler)
error-new-pc   (dc 1 error-interrupt-handler)
svc-new-pc     (dc 1 svc-interrupt-handler)
input-new-pc   (dc 1 input-interrupt-handler)
output-new-pc  (dc 1 output-interrupt-handler)
interrupt-data (dc 2 0)  ;; various interrupts cause information to be stored he
branch-address (dc 1 0)
time-slice     (dc 1 1000)
current-taskid (dc 1 0)
temp-r2        (dc 1 0)
temp-r3        (dc 1 0)
task-table     (dc 144 0)
segment-table  (dc 32 0)
readyq         (dc 20 0)
status-table   (dc 32 0)
ibuffers       (dc 128 0)
obuffers       (dc 128 0)
mbuffers       (dc 512 0)
               (dc 512 0)
               (dc 512 0)
               (dc 512 0)

;; ---------- KIT Source Code ----------
SAVE-STATE
(move  (2 temp-r2) (1 r2))             ;; Save R2
(move  (2 temp-r3) (1 r3))             ;; Save R3
(move  (1 r3) readyq)                  ;; R3 points to ready queue
(call  qfirst)                         ;; R2 has current task id
save-state-return
(mult  (1 r2) task-table-entry-length) ;; multiply by task table entry length
(add   (1 r2) task-table)              ;; R2 points to current task table entry
(move  (3 r2 pc-field)   (2 reg-save-area interrupt-pc-field))
(move  (3 r2 sp-field)   (2 reg-save-area interrupt-sp-field))
(move  (3 r2 r2-field)   (2 temp-r2))
(move  (3 r2 r3-field)   (2 temp-r3))
(move  (3 r2 r4-field)   (1 r4))
(move  (3 r2 r5-field)   (1 r5))
(move  (3 r2 r6-field)   (1 r6))
(move  (3 r2 r7-field)   (1 r7))
(add   (1 r2) flag-field)              ;; bump index register
(move  (3 r2)            (2 reg-save-area interrupt-flag-field))
(move  (1 r2)            (2 temp-r2))  ;; Restore R2 & R3.
(move  (1 r3)            (2 temp-r3))  ;; This is necessary for SVC interrupts.
```

```
(return)

RESTORE-STATE
;; Assume R2 has id of selected task.
(move  (1 r3) (1 r2))                  ;; R3 has next taskid, too
(mult  (1 r2) task-table-entry-length) ;; multiply by task table entry length
(add   (1 r2) task-table)              ;; R2 now points to the next task table e
(mult  (1 r3) 2)
(add   (1 r3) segment-table)           ;; R3 pts to segment table entry for next
(lbase  (3 r3 base-field))             ;; restore base register
(llimit (3 r3 limit-field))            ;; restore limit register
(move  (1 r3) (1 r2))
(add   (1 r3) flag-field)              ;; R3 points to flag field of task table
(move  (2 reg-save-area interrupt-pc-field)    (3 r2 pc-field))
(move  (2 reg-save-area interrupt-sp-field)    (3 r2 sp-field))
(move  (2 reg-save-area interrupt-flag-field)  (3 r3))
(move  (1 r7) (3 r2 r7-field))
(move  (1 r6) (3 r2 r6-field))
(move  (1 r5) (3 r2 r5-field))
(move  (1 r4) (3 r2 r4-field))
(move  (1 r3) (3 r2 r3-field))
(move  (1 r2) (3 r2 r2-field))
;; We must leave R0 & R1 alone since they're the PC & SP.
;; A LPSW will restore them from the register save area.
(return)

CLOCK-INTERRUPT-HANDLER
(call save-state)            ;; First, save the state of the current task.
trace-label1
(move (1 r3) readyq)         ;; R3 points to readyq
(call qfirst)                ;; Put current taskid in R2
(call dequeue)               ;; DEQUEUE the current task from the READYQ
(call enqueue)               ;; ENQUEUE the current task
trace-label2
(br dispatcher)              ;; Resume next task

ERROR-INTERRUPT-HANDLER
(call save-state)            ;; First, save the state of the current task.
trace-label3
(move (1 r3) readyq)         ;; R3 points to readyq
(call qfirst)                ;; Put current taskid in R2
(call dequeue)               ;; DEQUEUE the current task from the READYQ
trace-label4
(mult (1 r2) status-entry-length)
(add  (1 r2) status-table)   ;; r2 points to entry for current task in status ta
(move (3 r2 status-flag-field)    error-status)
(move (3 r2 status-taskid-field)  0)
(br dispatcher)              ;; Resume next task

SVC-INTERRUPT-HANDLER
;; The memory location SVCID-ADDR contains the svcid.
(call save-state)
trace-label5
(mod (2 svcid-addr) 4)       ;; Fix the svcid to a number less than 4.
(compare (2 svcid-addr) send-svcid)    ;; is it a request to SEND?
(brz  send-svc-handler)
(compare (2 svcid-addr) receive-svcid) ;; is it a request to RECEIVE?
(brz receive-svc-handler)
(compare (2 svcid-addr) tyo-svcid)   ;; a TYO request?
(brz tyo-svc-handler)
(br  tyi-svc-handler)
```

```
SEND-SVC-HANDLER
;; Conventions:
;;     Low order bits of R2 contain destination id
;;     R3 contains message.
;; Move these to R6 and R7.
(move (1 r6) (1 r2))
(mod  (1 r6) taskidlub)          ;; R6 has destination id
(move (1 r7) (1 r3))             ;; R7 has message
(move (1 r3) readyq)
(call qfirst)                    ;; R2 has current taskid
trace-label6
(move (2 current-taskid) (1 r2)) ;; save current taskid

;; Compute address of MBUFFER[source, dest], and test for a full buffer.
(move (1 r4) (1 r6))             ;; R4 contains destination id
(mult (1 r4) dest-multiplier)
(move (1 r3) (1 r2))             ;; R3 contains source id (i.e. current id)
(mult (1 r3) source-multiplier)
(add  (1 r3) (1 r4))
(add  (1 r3) mbuffers)           ;; R3 points to message buffer
trace-label7
(call qfullp)
(brz block-send)                 ;; If buffer full, block the sending task.

;; Else, message buffer isn't full. Perform send and resume task.
(move (1 r2) (1 r7))             ;; R2 has the message
(call enqueue)                   ;; R3 still points to the message buffer
trace-label8

;; Check for destination task waiting. R6 has destination taskid.
(move (1 r3) (1 r6))                ;; Move destination id to R3.
(mult (1 r3) status-entry-length)   ;; R3 has displacement to status entry
(add (1 r3)  status-table)          ;; R3 has absolute address of status entry
(compare (3 r3 status-flag-field) receive-status) ;; Waiting to receive?
(brnz svc-resume-task)              ;; If not, resume task.
(compare (3 r3 status-taskid-field) (2 current-taskid)) ;; Else, from current ta
(brnz svc-resume-task)              ;; If not, resume task
;; Else the destination task was waiting to receive from the current task.
;; Make it ready.
(move (3 r3 status-flag-field) ready-status)
(move (3 r3 status-taskid-field) 0)
(move (1 r2) (1 r6))                ;; R2 has destination id
(move (1 r3) readyq)                ;; R3 points to readyq
(call enqueue)
trace-label9
(br svc-resume-task)

BLOCK-SEND
;; Remove the current task from the readyq and mark it waiting to send.
(move (1 r3) readyq)
(call dequeue)
trace-label10
(move (1 r3) (2 current-taskid))
(mult (1 r3) status-entry-length)   ;; R3 has displacement to status entry
(add (1 r3)  status-table)          ;; R3 has absolute address of status entry
(move (3 r3 status-flag-field) send-status)
(move (3 r3 status-taskid-field) (1 r6))
(br dispatcher)

RECEIVE-SVC-HANDLER
;; Conventions:
```

```
;;     Low order bits of R2 contain source id
;;     Put message in R3 of current task.
(move (1 r6) (1 r2))
(mod  (1 r6) taskidlub)          ;; R6 has source id
(move (1 r3) readyq)
(call qfirst)                    ;; R2 has current taskid
trace-label11
(move (2 current-taskid) (1 r2)) ;; save current taskid

;; Compute address of MBUFFER[source, dest], and test for a full buffer.
(move (1 r4) (1 r2))             ;; R4 contains destination id (i.e. current id)
(mult (1 r4) dest-multiplier)
(move (1 r3) (1 r6))             ;; R3 contains source id
(mult (1 r3) source-multiplier)
(add  (1 r3) (1 r4))
(add  (1 r3) mbuffers)           ;; R3 points to message buffer
trace-label12
(call qemptyp)
(brz block-receive)              ;; If buffer empty, block the receiving task.

;; Else, message buffer isn't empty. Perform receive and resume task.
(call qfirst)                    ;; R2 has the message.
(call dequeue)                   ;; Dequeue the message buffer.
trace-label13
(move  (1 r3) (2 current-taskid))
(mult  (1 r3) task-table-entry-length) ;; multiply by task table entry length
(add   (1 r3) task-table)              ;; R3 points to current task table entry
(move  (3 r3 r3-field) (1 r2))         ;; Move message to current task's R3.
trace-label14

;; Check for source task waiting. R6 has source taskid.
(move (1 r3) (1 r6))               ;; Move source id to R3.
(mult (1 r3) status-entry-length)  ;; R3 has displacement to status entry
(add (1 r3)  status-table)         ;; R3 has absolute address of status entry
(compare (3 r3 status-flag-field) send-status) ;; Waiting to send?
(brnz svc-resume-task)             ;; If not, resume task.
(compare (3 r3 status-taskid-field) (2 current-taskid)) ;; Send to current task?
(brnz svc-resume-task)             ;; If not, resume task
;; Else the destination task was waiting to receive from the current task.
;; Make it ready.
(move (3 r3 status-flag-field) ready-status)
(move (3 r3 status-taskid-field) 0)
(move (1 r2) (1 r6))               ;; R2 has destination id
(move (1 r3) readyq)               ;; R3 points to readyq
(call enqueue)
trace-label15
(br svc-resume-task)

BLOCK-RECEIVE
;; Remove the current task from the readyq and mark it waiting to receive.
(move (1 r3) readyq)
(call dequeue)
trace-label16
(move (1 r3) (2 current-taskid))
(mult (1 r3) status-entry-length)  ;; R3 has displacement to status entry
(add (1 r3)  status-table)         ;; R3 has absolute address of status entry
(move (3 r3 status-flag-field) receive-status)
(move (3 r3 status-taskid-field) (1 r6))
(br dispatcher)

TYO-SVC-HANDLER
```

```
;; Conventions:
;;     Low order bits of R3 contain character.
;;     The current taskid is also the device id.
;; Move this to R7.
(move (1 r7) (1 r3))             ;; R7 has character
(move (1 r3) readyq)
(call qfirst)                    ;; R2 has current taskid
trace-label17
(move (2 current-taskid) (1 r2)) ;; save current taskid (equals device id)

;; Compute address of OBUFFER[devid], and test for a full buffer.
(move (1 r3) (1 r2))             ;; R3 contains devid (i.e. current taskid)
(mult (1 r3) output-buffer-length)
(add  (1 r3) obuffers)           ;; R3 points to the current output buffer
trace-label18
(call qfullp)
(brz block-tyo)                  ;; If buffer full, block the sending task.

;; Else, message buffer isn't full. Perform send and resume task.
(move (1 r2) (1 r7))             ;; R2 has the character.
(call enqueue)                   ;; R3 still points to the message buffer
trace-label19

;; Check for idle output device. If idle, post an output interrupt
(testo (2 current-taskid))  ;; Test for idle device
(brnz svc-resume-task)          ;; If not idle, resume task
(post (2 current-taskid))   ;; Else, post an output interrupt so the
;; output interrupt handler starts an output.
(br svc-resume-task)

BLOCK-TYO
;; Remove the current task from the readyq and mark it waiting to output.
(move (1 r3) readyq)
(call dequeue)
trace-label20
(move (1 r3) (2 current-taskid))
(mult (1 r3) status-entry-length)   ;; R3 has displacement to status entry
(add (1 r3)  status-table)          ;; R3 has absolute address of status entry
(move (3 r3 status-flag-field) output-status)
(move (3 r3 status-taskid-field) 0)
(br dispatcher)

TYI-SVC-HANDLER
;; Conventions:
;;     The current taskid is also the device id.
;; Put the input character in R3 of the current task.
(move (1 r3) readyq)
(call qfirst)                    ;; R2 has current taskid
trace-label21
(move (2 current-taskid) (1 r2)) ;; save current taskid (equals device id)

;; Compute address of IBUFFER[devid], and test for a empty buffer.
(move (1 r3) (1 r2))             ;; R3 contains devid (i.e. current taskid)
(mult (1 r3) input-buffer-length)
(add  (1 r3) ibuffers)           ;; R3 points to the current input buffer
trace-label22
(call qemptyp)
(brz block-tyi)                  ;; If buffer empty, block the current task.

;; Else, input buffer isn't empty. Perform input and resume task.
(call qfirst)                    ;; R2 has the next input character.
```

```
(call dequeue)                  ;; Dequeue the input buffer.
trace-label23
(move  (1 r3) (2 current-taskid))
(mult  (1 r3) task-table-entry-length) ;; multiply by task table entry length
(add   (1 r3) task-table)              ;; R3 points to current task table entry
(move  (3 r3 r3-field) (1 r2))         ;; Move message to current task's R3.
(br svc-resume-task)

BLOCK-TYI
;; Remove the current task from the readyq and mark it waiting to input.
(move (1 r3) readyq)
(call dequeue)
trace-label24
(move (1 r3) (2 current-taskid))
(mult (1 r3) status-entry-length)   ;; R3 has displacement to status entry
(add (1 r3)  status-table)          ;; R3 has absolute address of status entry
(move (3 r3 status-flag-field) input-status)
(move (3 r3 status-taskid-field) 0)
(br dispatcher)

INPUT-INTERRUPT-HANDLER
;; The memory location INPUT-DEVID-ADDR contains the ID of the interrupting devi
;; The memory location INPUT-CHAR-ADDR contains the input character.
;;
;; Pseudo Code:
;;
;; If the owning task is waiting to input
;;    then put the ID on the readyq
;;          update the status of the task
;; endif
;;
;; If the input buffer is full
;;    then replace the last queue element with the overflow character
;;    else if the input device signals an overflow error
;;            then enqueue an overflow character on the input buffer
;;            else  enqueue the character on the input buffer
;;         endif
;; endif
;;
(move (2 branch-address) dispatcher)   ;; initialize BRANCH-ADDRESS to DISPATCHE
                                       ;; exit via dispatcher when waiting
(move (2 temp-r3) (1 r3))              ;; Save R3 because we must use it.
(move (1 r3) readyq)                   ;; R3 points to readyq
(call qemptyp)                         ;; check for empty readyq;
                                       ;; if empty, no need to save state
trace-label25
(brz  iih-skip-save-state)
(move (2 branch-address) resume-task)  ;; We'll exit via RESUME-TASK
(move (1 r3) (2 temp-r3))              ;; Restore R3 for save-state
(call save-state)
trace-label26

iih-skip-save-state
(move (1 r5) (2 input-devid-addr))  ;; R5 has devid
(mult (1 r5) status-entry-length)   ;; R5 has displacement to status entry
(add (1 r5)  status-table)          ;; R5 has absolute address of status entry
(compare (3 r5 status-flag-field) input-status) ;; Waiting to input?
(brnz check-for-full-input-buffer)
;; The task which owns this device is waiting to input; Make it ready to run.
(move (1 r2) (2 input-devid-addr))  ;; R2 has taskid
(move (1 r3) readyq)                ;; R3 points to readyq
```

```
(call enqueue)                      ;; enq taskid on readyq
trace-label27
(move (3 r5 status-flag-field) ready-status)
(move (3 r5 status-taskid-field) 0)
trace-label28

check-for-full-input-buffer
(move (1 r3) (2 input-devid-addr))  ;; R3 has devid
(mult (1 r3) input-buffer-length)
(add  (1 r3) ibuffers)              ;; R3 points to the current input buffer
(call qfullp)
(brnz check-for-iport-error)
;; The input buffer is full. Replace the last queue element with the new charact
;; with the overflow bit set.
(move (1 r2) (2 input-char-addr))
(add  (1 r2) charlub)               ;; R2 now has character with the overflow bi
(call qreplace)                     ;; R3 still points to the current input buff
trace-label29
(br (2 branch-address))             ;; branch to either DISPATCHER or RESUME-TAS

check-for-iport-error
(testi (2 input-devid-addr))        ;; Test input device for overflow error
(brnz enqueue-input-character)      ;; if no error, enqueue the current characte
;; Else, enqueue the overflow character
(move (1 r2) (2 input-char-addr))
(add  (1 r2) charlub)               ;; R2 now has character with the overflow bi
(call enqueue)                      ;; R3 still points to the current input buff
trace-label30
(br (2 branch-address))             ;; branch to either DISPATCHER or RESUME-TAS

enqueue-input-character
(move (1 r2) (2 input-char-addr))
(call enqueue)                      ;; R3 still points to the current input buff
trace-label31
(br (2 branch-address))

OUTPUT-INTERRUPT-HANDLER
;; The location OUTPUT-DEVID-ADDR of memory contains id of the interrupting devi
;; This also happens to be the id of the process which owns that output device.
;;
;; Pseudo Code:
;;
;;  If  the owning task is waiting to output
;;      then put the id on the readyq
;;           update to status of the task
;;  endif
;;
;; if the output buffer is empty
;;    then clear the output interrupt
;;    else  start another output
;;          deq the output buffer
;; endif
;;
;; Resume the current task
;;
;; End of Pseudo Code
;;
(move (2 branch-address) dispatcher)   ;; initialize BRANCH-ADDRESS to DISPATCHE
                                       ;; exit via dispatcher when waiting
(move (2 temp-r3) (1 r3))              ;; Save R3 because we must use it.
(move (1 r3) readyq)                   ;; R3 points to readyq
```

```
(call qemptyp)                          ;; check for empty readyq;
                                        ;; if empty, no need to save state
trace-label32
(brz  oih-skip-save-state)
(move (2 branch-address) resume-task)  ;; We'll exit via RESUME-TASK
(move (1 r3) (2 temp-r3))              ;; Restore R3 for save-state
(call save-state)
trace-label33

oih-skip-save-state
(move (1 r5) (2 output-devid-addr)) ;; R5 has devid
(mult (1 r5) status-entry-length)   ;; R5 has displacement to status entry
(add (1 r5)  status-table)          ;; R5 has absolute address of status entry
(compare (3 r5 status-flag-field) output-status) ;; Waiting to output?
(brnz check-for-empty-output-buffer)
;; The task which owns this device is waiting to output; Make it ready to run.
(move (1 r2) (2 output-devid-addr)) ;; R2 has taskid
(move (1 r3) readyq)                ;; R3 points to readyq
(call enqueue)                      ;; enq taskid on readyq
trace-label34
(move (3 r5 status-flag-field) ready-status)
(move (3 r5 status-taskid-field) 0)
trace-label35

check-for-empty-output-buffer
(move (1 r3) (2 output-devid-addr)) ;; R3 has devid
(mult (1 r3) output-buffer-length)
(add  (1 r3) obuffers)              ;; R3 points to the current output buffer
(call qemptyp)
(brz  (2 branch-address))           ;; branch to either DISPATCHER or RESUME-TAS
;; Else the buffer is not empty, start the next output
(call qfirst)                       ;; Put the next output character in R2
(stout (2 output-devid-addr) (1 r2))
(call dequeue)                      ;; Deq the output buffer
trace-label36
(br (2 branch-address))             ;; branch to either DISPATCHER or RESUME-TAS

DISPATCHER ;; Allocate CPU to first task on readyq.
(move (1 r3) readyq)    ;; Point R3 to readyq
(call qemptyp)          ;; Readyq empty?
dispatcher-trace-label1
(brz  readyq-empty)
(call qfirst)           ;; Put next taskid in R2
(call restore-state)    ;; resume next task
dispatcher-trace-label2
(time (2 time-slice 0)) ;; set clock
(lpsw (2 reg-save-area))

READYQ-EMPTY
(wait)
pc-after-wait

SVC-RESUME-TASK ;; Return to current task (readyq is not empty).
(move (1 r3) readyq)    ;; Point R3 to readyq
(call qfirst)           ;; Put next taskid in R2
(call restore-state)    ;; resume next task
svc-resume-task-trace-label1
(svcr (2 reg-save-area))

RESUME-TASK ;; Return to current task (readyq is not empty).
(move (1 r3) readyq)    ;; Point R3 to readyq
```

```
(call qfirst)           ;; Put next taskid in R2
(call restore-state)    ;; resume next task
resume-task-trace-label1
(lpsw (2 reg-save-area))

ENQUEUE
;; Assume R2 contains item to enqueue
;;           R3 points to queue
;; this routine assumes queue not currently full
;; pseudo-code:
;;    store the item where ever the tail index points
;;    increment the current length
;;    increment the tail index (mod max-index)
(move  (1 r4) (1 r3))
(add   (1 r4) qarray-field)
(add   (1 r4) (3 r3 qtail-field))  ;; r4 has address of free slot
(move  (3 r4) (1 r2))              ;; store item
(incr  (3 r3 qcurrlength-field))   ;; increment current length
(incrm (3 r3 qtail-field) (3 r3 qmaxlength-field)) ;; increment tail
(return)

QREPLACE
;; Assume R3 points to non-empty queue.
;; Replace last queue element with contents of R2.
;;
(move  (1 r4) (3 r3 qtail-field))      ;; R4 has queue tail index
(decrm (1 r4) (3 r3 qmaxlength-field)) ;; decrement tail pointer
(add   (1 r4) (1 r3))                  ;; add address of queue
(add   (1 r4) qarray-field)            ;; R4 has address of last slot in queue
(move  (3 r4) (1 r2))                  ;; store item
(return)

DEQUEUE
;; assume R3 points to queue;
;; this routine assumes queue not currently empty
;; pseudo-code:
;;    decrement current queue length
;;    increment head index (mod maxlength)
(decr  (3 r3 qcurrlength-field))  ;; decrement the current length of the queue
(incrm (3 r3 qhead-field) (3 r3 qmaxlength-field))
(return)

QFIRST
;; Assume  R3 points to queue.
;; Put first queue item in R2.
;; This routine assumes queue not currently empty.
(move (1 r2) (1 r3))              ;; R2 points to queue
(add  (1 r2) qarray-field)        ;; R2 points to the qarray
(add  (1 r2) (3 r3 qhead-field))  ;; R2 points to the first queue element
(move (1 r2) (3 r2))              ;; put the first element into R2
(return)

QEMPTYP
;; assume R3 points to queue
;; set CC to zero if queue is empty
(compare (3 r3 qcurrlength-field) 0)
(return)

QFULLP
;; assume R3 points to queue
;; set CC to zero if queue is full
```

```
(compare (3 r3 qcurrlength-field) (3 r3 qmaxlength-field))
(return)

END-OF-OS-SOURCE
```

4.3 Flowcharts

As an aid to following the kernel, we present flowcharts for each interrupt handler. The flowcharts are not design aids, but were created after the fact to depict the control flow through each interrupt handler. There are 38 final states which can be reached after entering the kernel at the start of one of the interrupt handlers. These 38 final states are depicted by 38 exit boxes. Oval boxes are used to depict kernel entry and exit points. An oval box with a line beneath is a continuation onto a following page.

Chapter 5

The Verification of KIT

In this chapter we outline the correctness proof for the kernel. In Section OS-LAYER we define an interpreter **OS-PROCESSOR** which is intermediate between the target machine and abstract kernel. This machine captures the state transitions accomplished by the operating system implementation. We prove an equivalence theorem between a **TM-PROCESSOR** running KIT and an **OS-PROCESSOR**. The proof of **CORRECTNESS-OF-OPERATING-SYSTEM** (see Section 2.3) then reduces to proving that **OS-PROCESSOR** implements **AK-PROCESSOR**. Figure REVISED-PROOF-STRUCTURE is a modification of Figure 2-2 which reveals the role of **OS-PROCESSOR**. In subsequent sections of this chapter we discuss the proofs of **OS-IMPLEMENTS-AK** and **AK-IMPLEMENTS-PARALLEL-TASKS**.

5.1 The Operating System Layer

The operating system layer defines an interpreter which mediates between the target machine and the abstract kernel. It defines the transitions accomplished by KIT's interrupt handlers on the target machine. An operating system state is a **TM** loaded with a particular program. Therefore, the shell **TM** gives the structure of an **OS** state as well as a **TM** state. The predicate **GOOD-OS** defines the operating system layer state set and formalizes the pictorial description of the kernel layout given by Figure 4-1. **GOOD-OS** places constraints on various registers and memory locations.

We examine the conjuncts of **GOOD-OS**. First, an **OS** state must be a **GOOD-TM**. The next five of conjuncts of **GOOD-OS** define the contents of the interrupt vector. The predicate **(GOOD-CPU-LIST (TABLE (TM-CPU-LENGTH) (OS-TASK-TABLE OS)))** states that each entry of the task table is a valid CPU state. (We define the function **TABLE** below.) The next three

Figure 5-1: Revised KIT Proof Structure

conjuncts constrain the segment table. The segments defined by the segment table must all lie within main memory, they must be mutually disjoint and they must be disjoint from the kernel. The predicate **`(FINITE-NUMBER-QUEUEP (OS-READYQ OS) (AK-TASKIDLUB) (AK-TASKIDLUB))`** states that the ready queue is a bounded queue containing only valid task identifiers. The predicate **`(GOOD-STATUS-LIST (TABLE (AK-STATUS-LENGTH) (OS-STATUS-TABLE OS)))`** recognizes a valid status table implementation. The next three

conjuncts define valid implementations of the three buffer tables. The formula **(EQUAL (OS-CODE OS) (OS-MACHINE-CODE))** states that the code segment of the kernel contains a particular constant, the kernel machine code. The identity **(EQUAL (TM-SLIMIT OS) (OS-LIMIT))** requires the target machine slimit register to be equal to a particular number, defined by **(OS-LIMIT)**, large enough to contain the kernel. The predicate **(EQUAL (GETNTH (OS-TIME-SLICE-ADDRESS) (TM-MEMORY OS)) (AK-TIME-SLICE))** ensures that the time slice granted to tasks by the kernel is exactly the value specified by the abstract kernel.

```
Definition {1874}.
(GOOD-OS OS)
   =
(AND
 (GOOD-TM OS)
 (EQUAL (GETNTH (TM-CLOCK-NEW-PC-ADDR) (TM-MEMORY OS))
        (OS-CLOCK-INTERRUPT-HANDLER-ADDRESS))
 (EQUAL (GETNTH (TM-ERROR-NEW-PC-ADDR) (TM-MEMORY OS))
        (OS-ERROR-HANDLER-ADDRESS))
 (EQUAL (GETNTH (TM-SVC-NEW-PC-ADDR) (TM-MEMORY OS))
        (OS-SVC-HANDLER-ADDRESS))
 (EQUAL (GETNTH (TM-INPUT-NEW-PC-ADDR) (TM-MEMORY OS))
        (OS-INPUT-INTERRUPT-HANDLER-ADDRESS))
 (EQUAL (GETNTH (TM-OUTPUT-NEW-PC-ADDR) (TM-MEMORY OS))
        (OS-OUTPUT-INTERRUPT-HANDLER-ADDRESS))
 (GOOD-CPU-LIST (TABLE (TM-CPU-LENGTH) (OS-TASK-TABLE OS)))
 (FINITE-SEGMENT-TABLEP (TABLE 2 (OS-SEGMENT-TABLE OS)) (TM-MEMLENGTH))
 (MUTUALLY-DISJOINT (TABLE 2 (OS-SEGMENT-TABLE OS)))
 (DISJOINT-EVERYWHERE 0 (OS-LIMIT) (TABLE 2 (OS-SEGMENT-TABLE OS)))
 (FINITE-NUMBER-QUEUEP (OS-READYQ OS) (AK-TASKIDLUB) (AK-TASKIDLUB))
 (GOOD-STATUS-LIST (TABLE (AK-STATUS-LENGTH) (OS-STATUS-TABLE OS)))
 (FINITE-NUMBER-QUEUE-LISTP (TABLE (OS-IBUFFER-LENGTH) (OS-IBUFFERS OS))
                            (TASK-IBUFFER-CAPACITY)
                            (TM-WORDLUB))
 (FINITE-NUMBER-QUEUE-LISTP (TABLE (OS-OBUFFER-LENGTH) (OS-OBUFFERS OS))
                            (TASK-OBUFFER-CAPACITY)
                            (TM-WORDLUB))
 (FINITE-NUMBER-QUEUE-LISTP (TABLE (OS-MBUFFER-LENGTH) (OS-MBUFFERS OS))
                            (TASK-MBUFFER-CAPACITY)
                            (TM-WORDLUB))
 (EQUAL (OS-CODE OS) (OS-MACHINE-CODE))
 (EQUAL (TM-SLIMIT OS) (OS-LIMIT))
 (EQUAL (GETNTH (OS-TIME-SLICE-ADDRESS) (TM-MEMORY OS))
        (AK-TIME-SLICE))
 (NOT (TM-IN-SUPERVISOR-MODE OS))
 (PERMUTATION (MAPUP-QUEUE (OS-READYQ OS)) (OS-READY-SET OS))
 (IFF (TM-WAITING OS) (ARRAY-QEMPTYP (OS-READYQ OS)))
 (IMPLIES
  (NOT (TM-WAITING OS))
  (AND (EQUAL (TM-BASE OS)
              (BASE (GETNTH (OS-CURRENT-TASKID OS)
                            (TABLE 2 (OS-SEGMENT-TABLE OS)))))
       (EQUAL (TM-LIMIT OS)
              (LIMIT (GETNTH (OS-CURRENT-TASKID OS)
                             (TABLE 2 (OS-SEGMENT-TABLE OS))))))))
```

The remaining conjuncts of **GOOD-OS** define invariants on the operating system layer. First, the operating system interpreter is always in user mode. We next have two invariants that are present at the abstract kernel layer: the ready queue is a permutation of the set of ready tasks as defined by the status table, and the operating system is waiting if and only if the ready queue is empty. The final conjunct of **GOOD-OS** identifies the current base/limit register pair with a particular entry in the segment table.

The function **TABLE** referenced above is an abstraction function, which unflattens a flat representation of a table consisting of fixed-length elements of size **N**.

```
Definition {409}.
(TABLE N L)
   =
(IF (ZEROP N)
    L
    (IF (LISTP L)
        (CONS (GETSEG 0 N L)
              (TABLE N (NTHCDR N L)))
        NIL))
```

The function **OS-PROCESSOR** is the interpreter function for the operating system layer. It takes as arguments an **OS** state and an oracle which is identical to a **TM** oracle. **OS-STEP** is the single step function at the operating system layer. It defines an interrupt structure identical to **TM**'s. Recall that the state returned by **TM-STEP** on an interrupt is described by a simple PSW swap. The state returned by **OS-STEP** on an interrupt is not a PSW swap, but a machine state describing the effect of an interrupt handler. An **OS** interrupt step equals some positive number of **TM** steps occurring after the same interrupt.

```
Definition {3635}.
(OS-PROCESSOR OS ORACLE)
   =
(IF (LISTP ORACLE)
    (OS-PROCESSOR (OS-STEP (TM-POST-INTERRUPT (CAR ORACLE) OS))
                  (CDR ORACLE))
    OS)
```

```
Definition {3634}.
(OS-STEP OS)
   =
(IF (TM-INPUT-INTERRUPTP OS)
    (OS-INPUT-INTERRUPT-HANDLER OS)

(IF (TM-OUTPUT-INTERRUPTP OS)
    (OS-OUTPUT-INTERRUPT-HANDLER OS)

(IF (TM-WAITING OS)
    OS

(IF (TM-ERRORP OS)
    (OS-ERROR-HANDLER OS)

(IF (TM-CLOCK-INTERRUPTP OS)
    (OS-CLOCK-INTERRUPT-HANDLER OS)

(IF (TM-SVC-INTERRUPTP OS)
    (OS-SVC-HANDLER OS)

    (TM-FETCH-EXECUTE OS)))))))
```

We now provide more detailed information on the definitions of **GOOD-OS** and **OS-STEP** to make clear how the machine code program which defines KIT fits into the definition of the **OS** layer.

We examine the conjunct **(EQUAL (OS-CODE OS) (OS-MACHINE-CODE))** of **GOOD-OS** in some detail to see how **GOOD-OS** incorporates the assembled machine code into the definition of the **OS** layer. The function **OS-CODE** (see below) is defined to be a particular segment of memory. **(GETSEG N K L)** is the segment of list **L** beginning at location **N** with length **K**. The address and length of **OS-CODE** is determined by the values of particular labels in the symbol table constructed by the assembler. **OS-MACHINE-CODE** is that segment of the assembled source code which contains the machine code which we wish to have interpreted by the target machine. The value of **OS-MACHINE-CODE** is a list of numbers bounded by **TM-WORDLUB** which results from assembling the KIT source code. Other segments of memory which are mentioned in **GOOD-OS** are defined similarly.

```
Definition {1770}.
(OS-CODE OS)
   =
(GETSEG (OS-CODE-ADDRESS)
        (OS-CODE-LENGTH)
        (TM-MEMORY OS))

Definition {1751}.
(OS-CODE-ADDRESS) = (LOOKUP 'SAVE-STATE (OS-SYMTAB))
```

```
Definition {1755}.
(OS-CODE-LENGTH)
   =
(DIFFERENCE (LOOKUP 'END-OF-OS-SOURCE (OS-SYMTAB))
            (OS-CODE-ADDRESS))

Definition {1761}.
(OS-MACHINE-CODE)
   =
(GETSEG (OS-CODE-ADDRESS)
        (OS-CODE-LENGTH)
        (CAR (ASSEMBLE (OS-SOURCE))))
```

GOOD-OS constrains the target machine to be loaded with a particular program. The function **OS-STEP** gives the state changes produced by executing the program. We examine the clock interrupt handler in some detail. When in a state recognized by **GOOD-OS**, a clock interrupt causes the target machine to be placed in the supervisor mode, and places the address of the clock interrupt handler in the program counter. When in the supervisor mode **TM** is not interruptible. Therefore, **TM** will take some number of steps until the clock interrupt handler relinquishes control by resuming a task in user mode. The function **OS-CLOCK-INTERRUPT-HANDLER** defines the change to the state of the machine produced by the clock interrupt handler. (See the function **OS-STEP**.)

```
Definition {2288}.
(OS-CLOCK-INTERRUPT-HANDLER OS)
   =
(TM
 (PUTNTH
  (GETNTH (TIMES (TM-CPU-LENGTH)
                 (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS)))
          (OS-NEW-TASK-TABLE OS))
  0
  (PUTNTH
   (GETNTH (PLUS 1
                 (TIMES (TM-CPU-LENGTH)
                        (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS))))
           (OS-NEW-TASK-TABLE OS))
   1
   (PUTNTH
    (GETNTH (PLUS 8
                  (TIMES (TM-CPU-LENGTH)
                         (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS))))
            (OS-NEW-TASK-TABLE OS))
    2
    (PUTNTH (TM-R2 OS) (OS-TEMP-R2-ADDRESS)
     (PUTNTH (TM-R3 OS) (OS-TEMP-R3-ADDRESS)
      (PUTSEG (OS-NEW-TASK-TABLE OS) (OS-TASK-TABLE-ADDRESS)
       (PUTSEG (OS-SEGMENT-TABLE OS) (OS-SEGMENT-TABLE-ADDRESS)
        (PUTSEG (OS-CLOCK-NEW-READYQ OS) (OS-READYQ-ADDRESS)
         (PUTSEG (OS-CODE OS) (OS-CODE-ADDRESS)
          (PUTNTH (OS-SAVE-STATE-RETURN-ADDRESS)
                  (SUB1 (SUB1 (OS-LIMIT)))
           (PUTNTH (OS-DISPATCHER-TRACE-LABEL2) (SUB1 (OS-LIMIT))
                   (TM-MEMORY OS)))))))))))
 (OS-NEW-REGS (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS)) OS)
 (OS-NEW-CC (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS)) OS)
 (OS-NEW-ERROR (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS)) OS)
 (OS-NEW-SVCFLAG (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS)) OS)
 (OS-NEW-SVCID (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS)) OS)
 (OS-NEW-BASE (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS)) OS)
 (OS-NEW-LIMIT (ARRAY-QFIRST (OS-CLOCK-NEW-READYQ OS)) OS)
 (TM-SLIMIT OS)
 (TM-USER-MODE)
 (TM-RUN-STATE)
 (AK-TIME-SLICE)
 (TM-IPORTS OS)
 (TM-OPORTS OS))
```

OS-CLOCK-INTERRUPT-HANDLER is a large function. We let the theorem prover help us construct it as follows. We present to the theorem prover the event **TRACE-CLOCK-INTERRUPT-HANDLER** (see below), where **OS-INTENDED-CLOCK-INTERRUPT** defines the **TM** clock interrupt transition for a **GOOD-OS**, and **OS-TIME-FOR-CLOCK-INTERRUPT-HANDLER** is an oracle giving the number of steps required to complete execution of the clock interrupt handler (a list of ticks). Notice that the lemma states an equality which we do not expect to prove: that running the clock interrupt handler produces no state change.

```
Proposition.  TRACE-CLOCK-INTERRUPT-HANDLER (rewrite):
(IMPLIES
    (AND (GOOD-OS OS)
         (NOT (TM-WAITING OS)))
    (EQUAL (TM-PROCESSOR (OS-INTENDED-CLOCK-INTERRUPT OS)
                         (OS-TIME-FOR-CLOCK-INTERRUPT-HANDLER OS))
           OS))
```

In letting the theorem prover attempt a proof, the left hand side of the equality is rewritten to a form not involving calls to **TM-PROCESSOR** by replacing the call to **TM-PROCESSOR** with as many nested calls of **TM-STEP** as indicated by **OS-TIME-FOR-CLOCK-INTERRUPT-HANDLER.** The nested calls to **TM-STEP** are opened up and simplified. The resulting expression describes the final state of the clock interrupt handler. The rewriter in effect symbolically executes the operating system. We intercept the output of the theorem prover when the final state expression is generated, edit it to clean it up a bit, and use the resulting expression to define **OS-CLOCK-INTERRUPT-HANDLER.** We then submit the event **TRACE-CLOCK-INTERRUPT-HANDLER** again, this time placing the form **(OS-CLOCK-INTERRUPT-HANDLER OS)** on the right hand side of the equation.

```
Theorem {2296}.  TRACE-CLOCK-INTERRUPT-HANDLER (rewrite):
(IMPLIES
    (AND (GOOD-OS OS)
         (NOT (TM-WAITING OS)))
    (EQUAL (TM-PROCESSOR (OS-INTENDED-CLOCK-INTERRUPT OS)
                         (OS-TIME-FOR-CLOCK-INTERRUPT-HANDLER OS))
           (OS-CLOCK-INTERRUPT-HANDLER OS)))
```

The definition of **OS-STEP** is possible only when all paths through all interrupt handlers have been traced with such lemmas. The tracing lemmas and their support lemmas form a large part of the KIT script. The story sounds simple, but in fact the tracing lemmas are the most difficult to get the theorem prover to check. The lemmas require getting correct all the details of addressing complicated data structures.

The definition of **OS-STEP** handles the issue of time abstraction with respect to the correspondence of **TM** and **AK**. Figure TRACES compares the trace of a **TM** running KIT with a trace of an **OS** machine. In a **TM** trace, an interrupt step is followed by some number of fetch-execute steps occurring in supervisor mode. A contiguous number of such steps is accomplished in a single **OS** step. **OS-PROCESSOR** handles the time differential between **TM**

and **AK**. As a by-product of the definition of **OS-PROCESSOR**, we get termination of the operating system. **OS-PROCESSOR** can be defined only after running each path of the operating system to termination.

Figure 5-2: Traces of TM and OS

5.2 The Target Machine Implements the Operating System

In this section we discuss the equivalence of the target machine loaded with KIT and the operating system layer described in the previous section. The identity function is the abstraction function from a target machine state to an operating system state. **TM-PROCESSOR** and **OS-PROCESSOR** differ in the way they consume the oracle. As explained in Section 5.1, an interrupt step defined by **OS-STEP** comprehends multiple steps of the target machine. The exact relationship can be established by defining an intermediate processor **TIMED-TM-PROCESSOR**, which calls **TM-PROCESSOR** on each interrupt for as many steps as necessary to complete execution of the interrupt handler.

```
Definition {3639}.
(TIMED-TM-PROCESSOR TM ORACLE)
   =
(IF (LISTP ORACLE)
    (TIMED-TM-PROCESSOR
      (TIMED-TM-STEP (TM-POST-INTERRUPT (CAR ORACLE) TM))
      (CDR ORACLE))
    TM)
```

```
Definition {3638}.
(TIMED-TM-STEP TM)
   =
(IF (TM-INPUT-INTERRUPTP TM)
    (TM-PROCESSOR (TM-EXECUTE-INPUT-INTERRUPT TM)
                  (OS-TIME-FOR-INPUT-INTERRUPT-HANDLER TM))

(IF (TM-OUTPUT-INTERRUPTP TM)
    (TM-PROCESSOR (TM-EXECUTE-OUTPUT-INTERRUPT TM)
                  (OS-TIME-FOR-OUTPUT-INTERRUPT-HANDLER TM))

(IF (TM-WAITING TM)
    TM

(IF (TM-ERRORP TM)
    (TM-PROCESSOR (TM-EXECUTE-ERROR-INTERRUPT TM)
                  (OS-TIME-FOR-ERROR-HANDLER TM))

(IF (TM-CLOCK-INTERRUPTP TM)
    (TM-PROCESSOR (TM-EXECUTE-CLOCK-INTERRUPT TM)
                  (OS-TIME-FOR-CLOCK-INTERRUPT-HANDLER TM))

(IF (TM-SVC-INTERRUPTP TM)
    (TM-PROCESSOR (TM-EXECUTE-SVC-INTERRUPT TM)
                  (OS-TIME-FOR-SVC-HANDLER TM))

    (TM-FETCH-EXECUTE TM)))))))
```

Speaking loosely, **TIMED-TM-PROCESSOR** and **OS-PROCESSOR** run "faster" than **TM-PROCESSOR**. **TIMED-TM-PROCESSOR** and **OS-PROCESSOR** consume a single element of the oracle to handle an interrupt while **TM-PROCESSOR** requires more than one. The function **OS-ORACLE** constructs from an **OS-PROCESSOR** oracle an oracle with enough "ticks" inserted to enable **TM-PROCESSOR** to match the operation of **OS-PROCESSOR**.

```
Definition {3641}.
(OS-ORACLE OS ORACLE)
   =
(IF (LISTP ORACLE)
    (APPEND
      (OS-ORACLE-STEP (CAR ORACLE)
                      (TM-POST-INTERRUPT (CAR ORACLE) OS))
      (OS-ORACLE (TIMED-TM-STEP (TM-POST-INTERRUPT (CAR ORACLE) OS))
                 (CDR ORACLE)))
    ORACLE)
```

```
Definition {3640}.
(OS-ORACLE-STEP EVENT OS)
   =
(IF (TM-INPUT-INTERRUPTP OS)
    (CONS EVENT (OS-TIME-FOR-INPUT-INTERRUPT-HANDLER OS))

(IF (TM-OUTPUT-INTERRUPTP OS)
    (CONS EVENT (OS-TIME-FOR-OUTPUT-INTERRUPT-HANDLER OS))

(IF (TM-WAITING OS)
    (LIST EVENT)

(IF (TM-ERRORP OS)
    (CONS EVENT (OS-TIME-FOR-ERROR-HANDLER OS))

(IF (TM-CLOCK-INTERRUPTP OS)
    (CONS EVENT (OS-TIME-FOR-CLOCK-INTERRUPT-HANDLER OS))

(IF (TM-SVC-INTERRUPTP OS)
    (CONS EVENT (OS-TIME-FOR-SVC-HANDLER OS))

    (LIST EVENT)))))))
```

The lemma **TM-IMPLEMENTS-TIMED-TM** establishes the correspondence between **TM-PROCESSOR** and **TIMED-TM-PROCESSOR**. The interrupt branches of **TIMED-TM-PROCESSOR-STEP** have the form of the tracing lemmas used to generate the definition of **OS-STEP**. It therefore is a simple matter to prove that **TIMED-TM-PROCESSOR** is identical to **OS-PROCESSOR**, that is they describe the same function on a **GOOD-OS** state. We therefore get the *implements* relation between the **TM** and **OS** layers, stated in **TM-IMPLEMENTS-OS**. The theorem states that the **TM** layer implements the **OS** layer *if* I/O interrupts are adequately spaced - long enough to execute a path of the operating system. The longest path in our system takes 112 steps. So 112 is a crude measure of the minimum gap between I/O interrupts. This requirement carries up through higher layers of the proof.

```
Theorem {3644}.  TM-IMPLEMENTS-TIMED-TM (rewrite):
(EQUAL (TM-PROCESSOR TM (OS-ORACLE TM ORACLE))
       (TIMED-TM-PROCESSOR TM ORACLE))

Theorem {3654}.  TM-IMPLEMENTS-OS (rewrite):
(IMPLIES (GOOD-OS OS)
         (EQUAL (TM-PROCESSOR OS (OS-ORACLE OS ORACLE))
                (OS-PROCESSOR OS ORACLE)))
```

5.3 The Operating System Implements the Abstract Kernel

The proof that the operating system layer implements the abstract kernel is the heart of the verification of KIT. This result is established by the theorem **OS-IMPLEMENTS-AK**. Its proof is long. The abstraction function **MAPUP-OS** is large since there are many state components to map, and their mapping is non-trivial.

```
Theorem {4620}.  OS-IMPLEMENTS-AK (rewrite):
(IMPLIES (AND (GOOD-OS OS)
              (PLISTP ORACLE))
         (EQUAL (MAPUP-OS (OS-PROCESSOR OS ORACLE))
                (AK-PROCESSOR (MAPUP-OS OS) ORACLE)))
```

MAPUP-OS constructs an abstract kernel state from an operating system state. We will examine the mapping of each component, dispatching the simple ones first. Observe that the running/wait state flag, the program clock, the input ports and the output ports are mapped up to the abstract kernel with no transformation. The status table is mapped by the function **TABLE**, which collects a flat list into a list of tuples of a given size.

```
Definition {1953}.
(MAPUP-OS OS)
   =
(AK (MAPUP-OS-TASKS OS)
    (MAPUP-OS-IBUFFERS OS)
    (MAPUP-OS-OBUFFERS OS)
    (MAPUP-OS-MBUFFERS OS)
    (MAPUP-QUEUE (OS-READYQ OS))
    (TABLE (AK-STATUS-LENGTH)
           (OS-STATUS-TABLE OS))
    (TM-RWSTATE OS)
    (TM-CLOCK OS)
    (TM-IPORTS OS)
    (TM-OPORTS OS))
```

The mappings of the ready queue and buffer tables make use of a common abstraction function **MAPUP-QUEUE**, which maps an implementation of finite queues up to list structures. The formal details of **MAPUP-QUEUE** and a description of how we verify queue operations appears in Chapter QUEUES. Suffice it to say for the present that the operating system uses a circular implementation of finite queues.

```
Definition {1949}.
(MAPUP-QUEUE-LIST L)
   =
(IF (LISTP L)
    (CONS (MAPUP-QUEUE (CAR L))
          (MAPUP-QUEUE-LIST (CDR L)))
    NIL)
```

```
Definition {1950}.
(MAPUP-OS-IBUFFERS OS)
   =
(MAPUP-QUEUE-LIST (TABLE (OS-IBUFFER-LENGTH)
                         (OS-IBUFFERS OS)))

Definition {1951}.
(MAPUP-OS-OBUFFERS OS)
   =
(MAPUP-QUEUE-LIST (TABLE (OS-OBUFFER-LENGTH)
                         (OS-OBUFFERS OS)))

Definition {1952}.
(MAPUP-OS-MBUFFERS OS)
   =
(TABLE (AK-TASKIDLUB)
       (MAPUP-QUEUE-LIST (TABLE (OS-MBUFFER-LENGTH)
                                (OS-MBUFFERS OS))))
```

The function **MAPUP-OS-TASKS** maps out of the operating system state a list of task address spaces. An address space is defined as a target machine which contains just that portion of the machine which is visible to a single task running in user mode. The function **MAPUP-ADDRESS-SPACE** formalizes this notion. It builds a target machine which contains a given CPU state (general purpose registers and flags) and a segment of memory defined by a given base and limit. The base register is initialized to 0, and the limit register is initialized to the given limit. The machine is put in the user operating mode. Remaining target machine components have *don't care* values since they are not accessible in user mode.

```
Definition {1948}.
(MAPUP-OS-TASKS OS) = (MAPUP-TASKS 0 OS)

Definition {1947}.
(MAPUP-TASKS TASKID OS)
   =
(IF (LESSP TASKID (AK-TASKIDLUB))
    (CONS (MAPUP-TASK TASKID OS)
          (MAPUP-TASKS (ADD1 TASKID) OS))
    NIL)

Definition {1946}.
(MAPUP-TASK TASKID OS)
   =
(MAPUP-ADDRESS-SPACE (TM-MEMORY OS)
                     (MAPUP-REGS TASKID OS)
                     (MAPUP-CC TASKID OS)
                     (MAPUP-ERROR TASKID OS)
                     (MAPUP-SVCFLAG TASKID OS)
                     (MAPUP-SVCID TASKID OS)
                     (MAPUP-BASE TASKID OS)
                     (MAPUP-LIMIT TASKID OS))
```

```
Definition {1266}.
(MAPUP-ADDRESS-SPACE MEMORY REGS CC ERROR SVCFLAG SVCID BASE LIMIT)
   =
(TM (GETSEG BASE LIMIT MEMORY)
    REGS CC ERROR SVCFLAG SVCID 0 (FIX LIMIT)
    0 (TM-USER-MODE) 0 0 0 0)
```

The values chosen to initialize a task's address space (i.e. the values occurring as arguments to **MAPUP-ADDRESS-SPACE** in **MAPUP-TASK**) are extracted from the state of the operating system. The *i*th task's memory segment is defined by the segment of memory identified by the *i*th base/limit register pair in the segment table. The *i*th task's CPU state depends on whether or not the operating system is in the wait state, and the identity of the current task. If the operating system state is waiting, then the CPU state of task *i* is contained in the *i*th entry of the task table. If the operating system state is running, the CPU state of the current task is the current state of the CPU. The contents of the task table is not up to date for this task. Otherwise, the CPU state of the *i*th task is extracted from the task table. These points are formalized in the function **MAPUP-CPU**.

```
Definition {1939}.
(MAPUP-REGS TASKID OS)
   =
(GETSEG 0 (TM-REGLENGTH) (MAPUP-CPU TASKID OS))

Definition {1940}.
(MAPUP-CC TASKID OS)
   =
(TM-UNPACK-CC (GETNTH (TM-REGLENGTH) (MAPUP-CPU TASKID OS)))

Definition {1941}.
(MAPUP-ERROR TASKID OS)
   =
(TM-UNPACK-ERROR (GETNTH (TM-REGLENGTH) (MAPUP-CPU TASKID OS)))

Definition {1942}.
(MAPUP-SVCFLAG TASKID OS)
   =
(TM-UNPACK-SVCFLAG (GETNTH (TM-REGLENGTH) (MAPUP-CPU TASKID OS)))

Definition {1943}.
(MAPUP-SVCID TASKID OS)
   =
(TM-UNPACK-SVCID (GETNTH (TM-REGLENGTH) (MAPUP-CPU TASKID OS)))

Definition {1944}.
(MAPUP-BASE TASKID OS)
   =
(BASE (GETNTH TASKID (TABLE 2 (OS-SEGMENT-TABLE OS))))

Definition {1945}.
(MAPUP-LIMIT TASKID OS)
   =
(LIMIT (GETNTH TASKID (TABLE 2 (OS-SEGMENT-TABLE OS))))
```

```
Definition {1938}.
(MAPUP-CPU TASKID OS)
   =
(IF (TM-WAITING OS)
    (GETNTH TASKID
            (TABLE (TM-CPU-LENGTH)
                   (OS-TASK-TABLE OS)))
    (IF (EQUAL TASKID (OS-CURRENT-TASKID OS))
        (TM-CPU OS)
        (GETNTH TASKID
                (TABLE (TM-CPU-LENGTH)
                       (OS-TASK-TABLE OS)))))
```

We now can define a function required for the definition of the task layer, **GOOD-ADDRESS-SPACE**. This predicate must hold on the private state of a task. **GOOD-ADDRESS-SPACE** recognizes a TM with a memory of a given length, and running in user mode. **MAPUP-ADDRESS-SPACE** satisfies **GOOD-ADDRESS-SPACE** when it is constructed from a valid target machine.

```
Definition {1194}.
(GOOD-ADDRESS-SPACE X MEMLENGTH)
   =
(AND (TM-SHELLP X)
     (NUMBERP MEMLENGTH)
     (LEQ MEMLENGTH (TM-MEMLENGTH))
     (PLISTP (TM-MEMORY X))
     (FINITE-NUMBER-LISTP (TM-MEMORY X) (TM-WORDLUB))
     (EQUAL (LENGTH (TM-MEMORY X)) MEMLENGTH)
     (PLISTP (TM-REGS X))
     (FINITE-NUMBER-LISTP (TM-REGS X) (TM-WORDLUB))
     (EQUAL (LENGTH (TM-REGS X)) (TM-REGLENGTH))
     (FINITE-NUMBERP (TM-CC X) (TM-CCLUB))
     (FINITE-NUMBERP (TM-ERROR X) (TM-ERRORLUB))
     (FINITE-NUMBERP (TM-SVCFLAG X) (TM-SVCFLAGLUB))
     (FINITE-NUMBERP (TM-SVCID X) (TM-SVCIDLUB))
     (EQUAL (TM-BASE X) 0)
     (EQUAL (TM-LIMIT X) MEMLENGTH)
     (EQUAL (TM-SVMODE X) (TM-USER-MODE)))
```

The proof of **OS-IMPLEMENTS-AK** is by induction over the oracle argument to **AK-PROCESSOR**. It is a simple consequence of the theorem **OS-STEP-IMPLEMENTS-AK-STEP**, whose proof we give.

```
Theorem {4612}.  OS-STEP-IMPLEMENTS-AK-STEP (rewrite):
(IMPLIES (GOOD-OS OS)
         (EQUAL (MAPUP-OS (OS-STEP OS))
                (AK-STEP (MAPUP-OS OS))))

Proof:

This conjecture simplifies, rewriting with
OS-NOT-IN-SUPERVISOR-MODE, AK-SVC-INTERRUPTP-MAPUP-OS,
```

```
AK-CLOCK-INTERRUPTP-MAPUP-OS, AK-ERRORP-MAPUP-OS,
AK-WAITING-MAPUP-OS, AK-OUTPUT-INTERRUPTP-MAPUP-OS, and
AK-INPUT-INTERRUPTP-MAPUP-OS, and unfolding TM-OUTPUT-INTERRUPTP,
TM-INPUT-INTERRUPTP, OS-STEP, AK-INTERRUPTING-OUTPUT-PORT,
AK-INTERRUPTING-INPUT-PORT, and AK-STEP, to the following six new
formulas:

Case 6. (IMPLIES
         (AND (GOOD-OS OS)
              (NOT (TM-SOME-INPUT-INTERRUPTP (TM-IPORTS OS)))
              (TM-SOME-OUTPUT-INTERRUPTP (TM-OPORTS OS)))
         (EQUAL
          (MAPUP-OS (OS-OUTPUT-INTERRUPT-HANDLER OS))
          (AK-OUTPUT-INTERRUPT-HANDLER
             (TM-INTERRUPTING-OUTPUT-PORT (AK-OPORTS (MAPUP-OS OS)))
             (MAPUP-OS OS)))).

  But this again simplifies, applying the lemma
  CORRECTNESS-OF-OUTPUT-INTERRUPT-HANDLER, to:

        T.

Case 5. (IMPLIES
                (AND (GOOD-OS OS)
                     (NOT (TM-SOME-INPUT-INTERRUPTP (TM-IPORTS OS)))
                     (NOT (TM-SOME-OUTPUT-INTERRUPTP (TM-OPORTS OS)))
                     (NOT (TM-WAITING OS))
                     (TM-ERRORP OS))
                (EQUAL (MAPUP-OS (OS-ERROR-HANDLER OS))
                       (AK-ERROR-HANDLER (MAPUP-OS OS)))),

  which again simplifies, rewriting with
  CORRECTNESS-OF-OS-ERROR-HANDLER, to:

        T.

Case 4. (IMPLIES
                (AND (GOOD-OS OS)
                     (NOT (TM-SOME-INPUT-INTERRUPTP (TM-IPORTS OS)))
                     (NOT (TM-SOME-OUTPUT-INTERRUPTP (TM-OPORTS OS)))
                     (NOT (TM-WAITING OS))
                     (NOT (TM-ERRORP OS))
                     (NOT (TM-CLOCK-INTERRUPTP OS))
                     (TM-SVC-INTERRUPTP OS))
                (EQUAL (MAPUP-OS (OS-SVC-HANDLER OS))
                       (AK-SVC-HANDLER (MAPUP-OS OS)))).

  However this again simplifies, rewriting with the lemma
  CORRECTNESS-OF-SVC-HANDLER, to:

        T.

Case 3. (IMPLIES
                (AND (GOOD-OS OS)
                     (NOT (TM-SOME-INPUT-INTERRUPTP (TM-IPORTS OS)))
                     (NOT (TM-SOME-OUTPUT-INTERRUPTP (TM-OPORTS OS)))
                     (NOT (TM-WAITING OS))
                     (NOT (TM-ERRORP OS))
                     (NOT (TM-CLOCK-INTERRUPTP OS))
                     (NOT (TM-SVC-INTERRUPTP OS)))
                (EQUAL (MAPUP-OS (TM-FETCH-EXECUTE OS))
```

```
                    (AK-PRIVATE-STEP (MAPUP-OS OS)))),

  which again simplifies, applying the lemma
  CORRECTNESS-OF-TM-FETCH-EXECUTE, to:

        T.

Case 2. (IMPLIES
               (AND (GOOD-OS OS)
                    (NOT (TM-SOME-INPUT-INTERRUPTP (TM-IPORTS OS)))
                    (NOT (TM-SOME-OUTPUT-INTERRUPTP (TM-OPORTS OS)))
                    (NOT (TM-WAITING OS))
                    (NOT (TM-ERRORP OS))
                    (TM-CLOCK-INTERRUPTP OS))
               (EQUAL (MAPUP-OS (OS-CLOCK-INTERRUPT-HANDLER OS))
                      (AK-CLOCK-INTERRUPT-HANDLER (MAPUP-OS OS)))),

  which again simplifies, rewriting with
  CORRECTNESS-OF-CLOCK-INTERRUPT-HANDLER, to:

        T.

Case 1. (IMPLIES
         (AND (GOOD-OS OS)
              (TM-SOME-INPUT-INTERRUPTP (TM-IPORTS OS)))
         (EQUAL
          (MAPUP-OS (OS-INPUT-INTERRUPT-HANDLER OS))
          (AK-INPUT-INTERRUPT-HANDLER
              (TM-INTERRUPTING-INPUT-PORT (AK-IPORTS (MAPUP-OS OS)))
              (MAPUP-OS OS)))).

  This again simplifies, applying the lemma
  CORRECTNESS-OF-INPUT-INTERRUPT-HANDLER, to:

        T.

Q.E.D.
```

The lemmas **CORRECTNESS-OF-OUTPUT-INTERRUPT-HANDLER**, **CORRECTNESS-OF-INPUT-INTERRUPT-HANDLER**, **CORRECTNESS-OF-CLOCK-INTERRUPT-HANDLER**, **CORRECTNESS-OF-OS-ERROR-HANDLER** and **CORRECTNESS-OF-SVC-HANDLER** establish the correctness of each of the interrupt handlers and have identical form. The theorem **CORRECTNESS-OF-CLOCK-INTERRUPT-HANDLER** is stated as an example.

```
  Theorem {3680}.  CORRECTNESS-OF-CLOCK-INTERRUPT-HANDLER (rewrite):
  (IMPLIES (AND (GOOD-OS OS)
                (NOT (TM-WAITING OS))
                (NOT (TM-ERRORP OS)))
           (EQUAL (MAPUP-OS (OS-CLOCK-INTERRUPT-HANDLER OS))
                  (AK-CLOCK-INTERRUPT-HANDLER (MAPUP-OS OS))))
```

The proof of each interrupt handler correctness theorem follows the same pattern: open up the definition of **MAPUP-OS** and prove that the abstraction of each **OS** field

equals the corresponding **AK** field. The proof is therefore a large case split, the details of which we leave to the script.

The verification of the interrupt handlers gives five of the six cases required to prove **OS-STEP-IMPLEMENTS-AK-STEP**. The remaining case requires a proof that a fetch-execute step at the **OS** layer implements a fetch-execute step at the **AK** layer. This result is stated by the theorem **CORRECTNESS-OF-TM-FETCH-EXECUTE**.

```
[Theorem {2078}. CORRECTNESS-OF-TM-FETCH-EXECUTE (rewrite):
(IMPLIES (AND (GOOD-OS OS)
              (NOT (TM-WAITING OS))
              (NOT (TM-CLOCK-INTERRUPTP OS)))
         (EQUAL (MAPUP-OS (TM-FETCH-EXECUTE OS))
                (AK-FETCH-EXECUTE (MAPUP-OS OS))))
```

Recall that **AK-FETCH-EXECUTE** is defined as the application of **TM-FETCH-EXECUTE** to the current abstract address space. Therefore the proof of **CORRECTNESS-OF-TM-FETCH-EXECUTE** is a result about the interaction of **TM-FETCH-EXECUTE** and the address space abstraction. At the **AK** layer, address spaces are clearly isolated. Each address space is an element of the array **AK-PSTATES**. There is no sharing of data among address spaces. Therefore address space isolation is a simple result of the properties of array access. At the **OS** layer address space isolation is not nearly as transparent. A task's address space is computed from a segment of memory, the current CPU state and the current contents of the data structure **OS-TASK-TABLE**.

The proof of **CORRECTNESS-OF-TM-FETCH-EXECUTE** is accomplished by a case split on the current task identifier. The theorem **MAPUP-CURRENT-TASK-TM-FETCH-EXECUTE** states that **TM-FETCH-EXECUTE** behaves as desired on the current address space, and **MAPUP-TASK-SEPARATION** states the property that **TM-FETCH-EXECUTE** has no effect on an address space which is not current.

```
Theorem {2069}.  MAPUP-CURRENT-TASK-TM-FETCH-EXECUTE (rewrite):
(IMPLIES (AND (GOOD-OS OS)
              (NOT (TM-WAITING OS)))
         (EQUAL (MAPUP-TASK (OS-CURRENT-TASKID OS)
                            (TM-FETCH-EXECUTE OS))
                (TM-FETCH-EXECUTE (MAPUP-TASK (OS-CURRENT-TASKID OS)
                                              OS))))
```

```
Theorem {2070}.  MAPUP-TASK-SEPARATION (rewrite):
(IMPLIES (AND (GOOD-OS OS)
              (NOT (TM-WAITING OS))
              (NUMBERP TASKID)
              (LESSP TASKID (AK-TASKIDLUB))
              (NOT (EQUAL TASKID (OS-CURRENT-TASKID OS))))
         (EQUAL (MAPUP-TASK TASKID (TM-FETCH-EXECUTE OS))
                (MAPUP-TASK TASKID OS)))
```

These lemmas in turn rely on important properties of **TM**'s architecture. **TM-FETCH-EXECUTE-COMMUTES-WITH-MAPUP-ADDRESS-SPACE** states that **TM-FETCH-EXECUTE**, when in user mode, may be applied to the entire state of the target machine, or to just a single address space, with identical effect on that address space. It is this theorem which allows us to apply **TM-FETCH-EXECUTE** at all levels of the specification and definition of KIT. And it is this theorem which formalizes our intuitive understanding of what an address space is. The invariant defined by **GOOD-OS** ensures that the conditions required by this theorem always hold.

```
Theorem {1382}. TM-FETCH-EXECUTE-COMMUTES-WITH-MAPUP-ADDRESS-SPACE (rewr
(IMPLIES
   (AND (GOOD-TM TM)
        (LEQ (PLUS (TM-BASE TM) (TM-LIMIT TM))
             (TM-MEMLENGTH))
        (NOT (TM-IN-SUPERVISOR-MODE TM)))
   (EQUAL (TM-FETCH-EXECUTE (MAPUP-ADDRESS-SPACE (TM-MEMORY TM)
                                                 (TM-REGS TM)
                                                 (TM-CC TM)
                                                 (TM-ERROR TM)
                                                 (TM-SVCFLAG TM)
                                                 (TM-SVCID TM)
                                                 (TM-BASE TM)
                                                 (TM-LIMIT TM)))
          (MAPUP-ADDRESS-SPACE (TM-MEMORY (TM-FETCH-EXECUTE TM))
                               (TM-REGS (TM-FETCH-EXECUTE TM))
                               (TM-CC (TM-FETCH-EXECUTE TM))
                               (TM-ERROR (TM-FETCH-EXECUTE TM))
                               (TM-SVCFLAG (TM-FETCH-EXECUTE TM))
                               (TM-SVCID (TM-FETCH-EXECUTE TM))
                               (TM-BASE (TM-FETCH-EXECUTE TM))
                               (TM-LIMIT (TM-FETCH-EXECUTE TM))))))
```

The theorem **TM-FETCH-EXECUTE-MAPUP-ADDRESS-SPACE-SEPARATION** states the main protection theorem. In a machine state for which **GOOD-OS** holds, applying **TM-FETCH-EXECUTE** has no effect on the address space of a task which is not the current task.

```
Theorem {2048}.  TM-FETCH-EXECUTE-MAPUP-ADDRESS-SPACE-SEPARATION (rewrit
(IMPLIES
 (AND (GOOD-OS OS)
      (NOT (TM-WAITING OS))
      (NUMBERP TASKID)
      (LESSP TASKID (AK-TASKIDLUB))
      (NOT (EQUAL TASKID (OS-CURRENT-TASKID OS))))
 (EQUAL
     (MAPUP-ADDRESS-SPACE (TM-MEMORY (TM-FETCH-EXECUTE OS))
                          REGS CC ERROR SVCFLAG SVCID
                          (BASE (GETNTH TASKID
                                        (TABLE 2 (OS-SEGMENT-TABLE OS)))
                          (LIMIT (GETNTH TASKID
                                         (TABLE 2 (OS-SEGMENT-TABLE OS))
     (MAPUP-ADDRESS-SPACE (TM-MEMORY OS)
                          REGS CC ERROR SVCFLAG SVCID
                          (BASE (GETNTH TASKID
                                        (TABLE 2 (OS-SEGMENT-TABLE OS)))
                          (LIMIT (GETNTH TASKID
                                         (TABLE 2 (OS-SEGMENT-TABLE OS))
```

5.4 The Abstract Kernel Implements Tasks

The correctness theorem for the abstract kernel establishes that the kernel implements a set of independent tasks. The commutativity diagram in Figure AK-IMPLEMENTS-PARALLEL-TASKS depicts the relation which theorem **AK-IMPLEMENTS-PARALLEL-TASKS** states.

```
Theorem {1689}.  AK-IMPLEMENTS-PARALLEL-TASKS (rewrite):
(IMPLIES (AND (GOOD-AK AK)
              (FINITE-NUMBERP I (LENGTH (AK-PSTATES AK))))
         (EQUAL (PROJECT I (AK-PROCESSOR AK ORACLE))
                (TASK-PROCESSOR (PROJECT I AK)
                                I
                                (CONTROL-ORACLE I AK ORACLE))))
```

The abstraction function is **PROJECT**, which projects the state of the *i*th task out of an abstract kernel state. **PROJECT** composes a task state from the *i*th address space and the shared buffers.

```
Definition {1672}.
(PROJECT I AK)
   =
(TASK (GETNTH I (AK-PSTATES AK))
      (AK-CHANNELS AK))

Definition {1671}.
(AK-CHANNELS AK)
   =
(LIST (AK-IBUFFERS AK) (AK-OBUFFERS AK) (AK-MBUFFERS AK))
```

In **AK-IMPLEMENTS-PARALLEL-TASKS**, **TASK-PROCESSOR**'s oracle is a function of the

Figure 5-3: AK Implements Parallel Tasks

task identifier, the initial abstract kernel state and the abstract kernel's oracle. **CONTROL-ORACLE** mirrors **AK-PROCESSOR**. It constructs an oracle for the task layer by building a list which at each step contains **T** if the indicated task is current in the abstract kernel, or contains the shared state which results from a step of the abstract kernel.

```
Definition {1674}.
(CONTROL-ORACLE I AK ORACLE)
   =
(IF (LISTP ORACLE)
    (CONS
     (CONTROL-ORACLE-STEP I (AK-POST-INTERRUPT (CAR ORACLE) AK))
     (CONTROL-ORACLE I
                     (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                     (CDR ORACLE)))
    NIL)
```

```
Definition {1673}.
(CONTROL-ORACLE-STEP I AK)
   =
(IF (AK-INPUT-INTERRUPTP AK)
    (AK-CHANNELS
      (AK-INPUT-INTERRUPT-HANDLER
        (TM-INTERRUPTING-INPUT-PORT (AK-IPORTS AK))
        AK))

(IF (AK-OUTPUT-INTERRUPTP AK)
    (AK-CHANNELS
      (AK-OUTPUT-INTERRUPT-HANDLER
        (TM-INTERRUPTING-OUTPUT-PORT (AK-OPORTS AK))
        AK))

(IF (AK-WAITING AK)
    (AK-CHANNELS AK)

(IF (AK-ERRORP AK)
    (AK-CHANNELS (AK-ERROR-HANDLER AK))

(IF (AK-CLOCK-INTERRUPTP AK)
    (AK-CHANNELS (AK-CLOCK-INTERRUPT-HANDLER AK))

(IF (AK-SVC-INTERRUPTP AK)
    (IF (EQUAL I (AK-TASKID AK))
        T
        (AK-CHANNELS (AK-SVC-HANDLER AK)))

    (IF (EQUAL I (AK-TASKID AK))
        T
        (AK-CHANNELS (AK-PRIVATE-STEP AK)))))))))
```

The proof of **AK-IMPLEMENTS-PARALLEL-TASKS** is by induction on **ORACLE** and is given below. The induction step, **CASE 2**, is proved by a case split which considers whether or not the task identifier **I** indicates an active task. We must have that an AK step implements a step on an active task as specified by the task layer. On a non-active task, the control oracle constructed by **CONTROL-ORACLE-STEP** must contain the shared state which **AK** generates.

```
Theorem {1689}.  AK-IMPLEMENTS-PARALLEL-TASKS (rewrite):
(IMPLIES (AND (GOOD-AK AK)
              (FINITE-NUMBERP I (LENGTH (AK-PSTATES AK))))
         (EQUAL (PROJECT I (AK-PROCESSOR AK ORACLE))
                (TASK-PROCESSOR (PROJECT I AK)
                                I
                                (CONTROL-ORACLE I AK ORACLE))))

Proof:

This conjecture can be simplified, using the abbreviations
FINITE-NUMBERP, IMPLIES, NOT, OR, AND, ACCESS-AK-POST-INTERRUPT,
and LENGTH-AK-PSTATES-AK-STEP, to two new formulas:
```

```
Case 2. (IMPLIES
         (AND
          (LISTP ORACLE)
          (IMPLIES
           (AND
             (GOOD-AK (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK)))
             (FINITE-NUMBERP I
                             (LENGTH (AK-PSTATES AK))))
           (EQUAL
            (PROJECT I
             (AK-PROCESSOR
                       (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                       (CDR ORACLE)))
            (TASK-PROCESSOR
             (PROJECT I
                      (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK)))
             I
             (CONTROL-ORACLE I
                       (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                       (CDR ORACLE)))))
          (GOOD-AK AK)
          (NUMBERP I)
          (LESSP I (LENGTH (AK-PSTATES AK))))
         (EQUAL (PROJECT I (AK-PROCESSOR AK ORACLE))
                (TASK-PROCESSOR (PROJECT I AK)
                                I
                                (CONTROL-ORACLE I AK ORACLE)))),
```

which simplifies, applying GOOD-AK-AK-POST-INTERRUPT, GOOD-AK-AK-STEP, CDR-CONS, and CAR-CONS, and opening up FINITE-NUMBERP, AND, IMPLIES, AK-PROCESSOR, CONTROL-ORACLE, and TASK-PROCESSOR, to the following two new formulas:

```
Case 2.2.
         (IMPLIES
          (AND
           (LISTP ORACLE)
           (EQUAL
            (PROJECT I
             (AK-PROCESSOR
                       (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                       (CDR ORACLE)))
            (TASK-PROCESSOR
             (PROJECT I
                      (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK)))
             I
             (CONTROL-ORACLE I
                       (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                       (CDR ORACLE))))
           (GOOD-AK AK)
           (NUMBERP I)
           (LESSP I (LENGTH (AK-PSTATES AK)))
           (NOT
            (TASK-ACTIVEP
               (CONTROL-ORACLE-STEP I
                                    (AK-POST-INTERRUPT (CAR ORACLE)
                                                       AK)))))
          (EQUAL
           (PROJECT I
            (AK-PROCESSOR
                       (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
```

```
                (CDR ORACLE)))
          (TASK-PROCESSOR
           (TASK-UPDATE-SHARED-STATE
            (PROJECT I AK)
            (CONTROL-ORACLE-STEP I
                             (AK-POST-INTERRUPT (CAR ORACLE) AK)))
           I
           (CONTROL-ORACLE I
                (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                (CDR ORACLE))))).
```

This again simplifies, using linear arithmetic, rewriting with the lemma AK-IMPLEMENTS-NON-ACTIVE-TASK-STEP, and expanding the function FINITE-NUMBERP, to:

```
      T.
```

```
Case 2.1.
        (IMPLIES
         (AND
          (LISTP ORACLE)
          (EQUAL
           (PROJECT I
            (AK-PROCESSOR
                (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                (CDR ORACLE)))
           (TASK-PROCESSOR
            (PROJECT I
                   (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK)))
            I
            (CONTROL-ORACLE I
                (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                (CDR ORACLE))))
          (GOOD-AK AK)
          (NUMBERP I)
          (LESSP I (LENGTH (AK-PSTATES AK)))
          (TASK-ACTIVEP
           (CONTROL-ORACLE-STEP I
                            (AK-POST-INTERRUPT (CAR ORACLE) AK))))
         (EQUAL
          (PROJECT I
           (AK-PROCESSOR
                (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                (CDR ORACLE)))
          (TASK-PROCESSOR
           (TASK-STEP (PROJECT I AK) I)
           I
           (CONTROL-ORACLE I
                (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
                (CDR ORACLE))))),
```

which again simplifies, using linear arithmetic, applying the lemma AK-IMPLEMENTS-ACTIVE-TASK-STEP, and expanding FINITE-NUMBERP, to:

```
      T.
```

```
Case 1. (IMPLIES
            (AND (NOT (LISTP ORACLE))
                 (GOOD-AK AK)
                 (NUMBERP I)
```

```
                (LESSP I (LENGTH (AK-PSTATES AK))))
           (EQUAL (PROJECT I (AK-PROCESSOR AK ORACLE))
                  (TASK-PROCESSOR (PROJECT I AK)
                                  I
                                  (CONTROL-ORACLE I AK ORACLE)))),
```

which simplifies, opening up the functions AK-PROCESSOR, CONTROL-ORACLE, LISTP, and TASK-PROCESSOR, to:

```
      T.

Q.E.D.
```

The critical support lemmas used in **AK-IMPLEMENTS-PARALLEL-TASKS** are given below. **AK-IMPLEMENTS-ACTIVE-TASK-STEP** establishes that the abstract kernel's transition on the current task is identical to a step on an active task at the task layer. This requires checking that the communication primitives are implemented correctly, which is not difficult since the representation of buffers is identical at the abstract kernel and task layers. The lemma **AK-IMPLEMENTS-NON-ACTIVE-TASK-STEP** is a matter of demonstrating that **CONTROL-ORACLE** contains the appropriate shared state on a non-active task step, and that no transition occurs on the indicated task's private state.

```
Theorem {1688}.  AK-IMPLEMENTS-ACTIVE-TASK-STEP (rewrite):
(IMPLIES
 (AND (GOOD-AK AK)
      (FINITE-NUMBERP I (LENGTH (AK-PSTATES AK)))
      (TASK-ACTIVEP
        (CONTROL-ORACLE-STEP I
                             (AK-POST-INTERRUPT (CAR ORACLE) AK))))
 (EQUAL (PROJECT I (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK)))
        (TASK-STEP (PROJECT I AK) I)))

Theorem {1681}.  AK-IMPLEMENTS-NON-ACTIVE-TASK-STEP (rewrite):
(IMPLIES
 (AND (GOOD-AK AK)
      (FINITE-NUMBERP I (LENGTH (AK-PSTATES AK)))
      (NOT (TASK-ACTIVEP
             (CONTROL-ORACLE-STEP I
                                  (AK-POST-INTERRUPT (CAR ORACLE)
                                                     AK)))))
 (EQUAL (PROJECT I (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK)))
        (TASK-UPDATE-SHARED-STATE
          (PROJECT I AK)
          (CONTROL-ORACLE-STEP I
                               (AK-POST-INTERRUPT (CAR ORACLE)
                                                  AK))))))
```

5.5 Composing the Interpreter Equivalence Theorems

We have described the correctness proof between each consecutive pair of layers in Figure 5-1. These lemmas can be used to prove a single theorem which spans all layers. Using **TM-IMPLEMENTS-OS** and **OS-IMPLEMENTS-AK** we get the theorem **CORRECTNESS-OF-OPERATING-SYSTEM** which is the main operating system correctness theorem. Recall from Section 5.2 that the theorem states that the **TM** layer matches the **AK** layer if I/O interrupts occur at the **TM** layer with a frequency low enough to always allow an interrupt handler to complete. The longest time span taken by an interrupt handler in KIT is 112 steps, so this is a crude measure of the minimum gap between I/O interrupts. This figure is fairly small, so the frequency of interrupts is not required to be very low. If the frequency condition is violated at the **TM** layer, then an interrupt will be ignored, and process isolation will still be preserved.

```
Theorem {4621}.  CORRECTNESS-OF-OPERATING-SYSTEM (rewrite):
(IMPLIES (AND (GOOD-OS OS)
              (PLISTP ORACLE))
         (EQUAL (MAPUP-OS (TM-PROCESSOR OS (OS-ORACLE OS ORACLE)))
                (AK-PROCESSOR (MAPUP-OS OS) ORACLE)))
```

Combining this result with **AK-IMPLEMENTS-PARALLEL-TASKS** gives the result which spans the entire ladder of Figure 5-1. The theorem **OS-IMPLEMENTS-PARALLEL-TASKS** establishes the result that the operating system running on the target machine **TM** implements our abstract definition of a parallel process.

```
Theorem {4623}.  OS-IMPLEMENTS-PARALLEL-TASKS:
(IMPLIES (AND (GOOD-OS OS)
              (PLISTP ORACLE)
              (FINITE-NUMBERP I (LENGTH (AK-PSTATES (MAPUP-OS OS)))))
         (EQUAL (PROJECT-ITH-TASK I
                                  (TM-PROCESSOR OS
                                                (OS-ORACLE OS ORACLE)))
                (TASK-PROCESSOR (PROJECT-ITH-TASK I OS)
                                I
                                (CONTROL-ORACLE I
                                                (MAPUP-OS OS)
                                                ORACLE))))
```

Chapter 6

Queues

In this chapter we take a detailed look at how we verify operations on queues. The buffers and ready queue of the abstract kernel are implemented as bounded queues, so this explanation reveals much of the effort involved in proving that the operating system implements the abstract kernel.

6.1 An Implementation of Queues

We have already seen the queue primitives at the abstract kernel level. We repeat them here.

```
Definition {470}.
(QFIRST TABLE) = (CAR TABLE)

Definition {471}.
(ENQ ITEM TABLE) = (APPEND TABLE (LIST ITEM))

Definition {472}.
(DEQ TABLE) = (CDR TABLE)

Definition {473}.
(QEMPTYP TABLE) = (EQUAL (LENGTH TABLE) 0)

Definition {474}.
(QFULLP TABLE MAX) = (NOT (LESSP (LENGTH TABLE) MAX))

Definition {475}.
(QREPLACE ITEM QUEUE) = (ENQ ITEM (NONLAST QUEUE))

Definition {458}.
(NONLAST L) = (GETSEG 0 (SUB1 (LENGTH L)) L)
```

We wish to implement queues of finite length in the store of a computer. We choose to implement them circularly. That is, a head and tail pointer cycle around a fixed size segment of memory. We call our queue implementation an *array queue*, suggesting an implementation in a memory array. We implement queues whose contents are single memory words.

The format of an array queue is a 4-tuple appended to a memory segment containing the queue elements. We give the format of the 4-tuple, where **QARRAY** refers to the appended memory segment.

- **HEAD** : An index into **QARRAY** giving the location of the first queue element.
- **TAIL** : An index into **QARRAY** giving the location of the first free slot at the end of the queue.
- **CURRLENGTH** : The current length of the queue.
- **MAXLENGTH** : The maximum length of the queue. The length of **QARRAY**.

The format of an **ARRAY-QUEUE** is formalized by the following definitions, which give indices to the various queue fields within an **ARRAY-QUEUE**.

```
Definition {494}.
(QHEAD-FIELD) = 0

Definition {495}.
(QTAIL-FIELD) = 1

Definition {496}.
(QCURRLENGTH-FIELD) = 2

Definition {497}.
(QMAXLENGTH-FIELD) = 3

Definition {498}.
(QARRAY-FIELD) = 4
```

The predicate **ARRAY-QUEUEP** recognizes a segment of memory which contains a well formed **ARRAY-QUEUE**. It states all the required relationships among the fields of a queue. The most intricate property is expressed by the function **ARRAY-QINDEX-RELATION** which relates the **HEAD** and **TAIL** positions to the current length of the queue. Figure DELTA depicts the measurement made by the function **DELTA**, the "wrap around" distance from **HEAD** to **TAIL** in a queue. **ARRAY-QINDEX-RELATION** states that if **HEAD** and **TAIL** are identical, then the queue length is **QMAXLENGTH** (i.e. the queue is full), otherwise the queue length is **QCURRLENGTH**.

Figure 6-1: Delta

```
Definition {506}.
(ARRAY-QUEUEP QUEUE)
   =
(AND (PLISTP QUEUE)
     (EQUAL (LENGTH QUEUE)
            (PLUS (QARRAY-FIELD)
                  (GETNTH (QMAXLENGTH-FIELD) QUEUE)))
     (NUMBERP (GETNTH (QHEAD-FIELD) QUEUE))
     (NUMBERP (GETNTH (QTAIL-FIELD) QUEUE))
     (NUMBERP (GETNTH (QCURRLENGTH-FIELD) QUEUE))
     (NOT (ZEROP (GETNTH (QMAXLENGTH-FIELD) QUEUE)))
     (LESSP (GETNTH (QHEAD-FIELD) QUEUE)
            (GETNTH (QMAXLENGTH-FIELD) QUEUE))
     (LESSP (GETNTH (QTAIL-FIELD) QUEUE)
            (GETNTH (QMAXLENGTH-FIELD) QUEUE))
     (LESSP (GETNTH (QCURRLENGTH-FIELD) QUEUE)
            (ADD1 (GETNTH (QMAXLENGTH-FIELD) QUEUE)))
     (ARRAY-QINDEX-RELATION QUEUE))

Definition {505}.
(ARRAY-QINDEX-RELATION QUEUE)
   =
(EQUAL (DELTA (GETNTH (QHEAD-FIELD) QUEUE)
              (GETNTH (QTAIL-FIELD) QUEUE)
              (GETNTH (QMAXLENGTH-FIELD) QUEUE))
       (IF (ZEROP (GETNTH (QCURRLENGTH-FIELD) QUEUE))
           (GETNTH (QMAXLENGTH-FIELD) QUEUE)
           (GETNTH (QCURRLENGTH-FIELD) QUEUE)))

Definition {499}.
(DELTA A B MAX)
   =
(IF (LEQ B A)
    (PLUS (DIFFERENCE MAX A) B)
    (DIFFERENCE B A))
```

The following functions define **ARRAY-QUEUE** primitives which correspond to the abstract queue primitives. They state precisely how the state of an **ARRAY-QUEUE** is changed by each operation. Recall that the form **(PUTNTH V N L)** is the list which is identical to **L** except for the **N**th element, which is equal to **V**. **(GETNTH N L)** is the **N**th element of **L**.

```
Definition {520}.
(ARRAY-ENQ ITEM QUEUE)
   =
(PUTNTH (INCR-MOD (GETNTH (QTAIL-FIELD) QUEUE)
                  (GETNTH (QMAXLENGTH-FIELD) QUEUE))
        (QTAIL-FIELD)
        (PUTNTH (ADD1 (GETNTH (QCURRLENGTH-FIELD) QUEUE))
                (QCURRLENGTH-FIELD)
                (PUTNTH ITEM
                        (PLUS (QARRAY-FIELD)
                              (GETNTH (QTAIL-FIELD) QUEUE))
                        QUEUE)))

Definition {521}.
(ARRAY-DEQ QUEUE)
   =
(PUTNTH (INCR-MOD (GETNTH (QHEAD-FIELD) QUEUE)
                  (GETNTH (QMAXLENGTH-FIELD) QUEUE))
        (QHEAD-FIELD)
        (PUTNTH (SUB1 (GETNTH (QCURRLENGTH-FIELD) QUEUE))
                (QCURRLENGTH-FIELD)
                QUEUE))

Definition {522}.
(ARRAY-QFIRST QUEUE)
   =
(GETNTH (PLUS (QARRAY-FIELD)
              (GETNTH (QHEAD-FIELD) QUEUE))
        QUEUE)

Definition {523}.
(ARRAY-QFULLP QUEUE)
   =
(EQUAL (GETNTH (QCURRLENGTH-FIELD) QUEUE)
       (GETNTH (QMAXLENGTH-FIELD) QUEUE))

Definition {524}.
(ARRAY-QEMPTYP QUEUE)
   =
(ZEROP (GETNTH (QCURRLENGTH-FIELD) QUEUE))

Definition {560}.
(ARRAY-QREPLACE ITEM QUEUE)
   =
(ARRAY-ENQ ITEM (ARRAY-NONLAST QUEUE))

Definition {547}.
(ARRAY-NONLAST QUEUE)
   =
(PUTNTH (DECR-MOD (GETNTH (QTAIL-FIELD) QUEUE)
                  (GETNTH (QMAXLENGTH-FIELD) QUEUE))
        (QTAIL-FIELD)
        (PUTNTH (SUB1 (GETNTH (QCURRLENGTH-FIELD) QUEUE))
                (QCURRLENGTH-FIELD)
                QUEUE))

Definition {464}.
(INCR-MOD N LUB)
   =
(IF (LESSP (ADD1 N) LUB) (ADD1 N) 0)

Definition {465}.
(DECR-MOD N LUB)
   =
(IF (ZEROP N) (SUB1 LUB) (SUB1 N))
```

6.2 The Correctness of the Queue Implementation

MAPUP-QUEUE is the abstraction function from **ARRAY-QUEUE**s to abstract queues. **DELTA-SEGMENT** is the workhorse. In the definition of **DELTA-SEGMENT**, **S** is a list and **A** and **B** are indices into the list. **DELTA-SEGMENT** constructs the segment which wraps around **S**, whose first element is **S(A)** and whose last element is the one preceding **S(B)**.

```
Definition {579}.
(MAPUP-QUEUE QUEUE)
   =
(IF (ARRAY-QEMPTYP QUEUE)
    NIL
    (DELTA-SEGMENT (GETNTH (QHEAD-FIELD) QUEUE)
                   (GETNTH (QTAIL-FIELD) QUEUE)
                   (GETSEG (QARRAY-FIELD)
                           (GETNTH (QMAXLENGTH-FIELD) QUEUE)
                           QUEUE)))
```

```
Definition {574}.
(DELTA-SEGMENT A B S)
   =
(IF (LEQ B A)
    (APPEND (GETSEG A (DIFFERENCE (LENGTH S) A) S)
            (GETSEG 0 B S))
    (GETSEG A (DIFFERENCE B A) S))
```

The relationship between **DELTA** and **DELTA-SEGMENT** is expressed by the theorem **DELTA-EQUALS-LENGTH-DELTA-SEGMENT.**

```
Theorem {575}.  DELTA-EQUALS-LENGTH-DELTA-SEGMENT (rewrite):
(IMPLIES (AND (LESSP A (LENGTH S))
              (LESSP B (LENGTH S)))
         (EQUAL (LENGTH (DELTA-SEGMENT A B S))
                (DELTA A B (LENGTH S)))).
```

We prove the correctness of the **ARRAY-QUEUE** implementation before we consider any machine code. The following theorems establish the correctness of each of the queue primitives.

```
Theorem {585}.  CORRECTNESS-OF-ARRAY-ENQ (rewrite):
(IMPLIES (AND (ARRAY-QUEUEP QUEUE)
              (NOT (ARRAY-QFULLP QUEUE)))
         (EQUAL (MAPUP-QUEUE (ARRAY-ENQ ITEM QUEUE))
                (ENQ ITEM (MAPUP-QUEUE QUEUE)))).
```

```
Theorem {593}.  CORRECTNESS-OF-ARRAY-DEQ (rewrite):
(IMPLIES (AND (ARRAY-QUEUEP QUEUE)
              (NOT (ARRAY-QEMPTYP QUEUE)))
         (EQUAL (MAPUP-QUEUE (ARRAY-DEQ QUEUE))
                (DEQ (MAPUP-QUEUE QUEUE)))).
```

```
Theorem {594}.  CORRECTNESS-OF-ARRAY-QFIRST (rewrite):
(IMPLIES (AND (ARRAY-QUEUEP QUEUE)
              (NOT (ARRAY-QEMPTYP QUEUE)))
         (EQUAL (ARRAY-QFIRST QUEUE)
                (QFIRST (MAPUP-QUEUE QUEUE))))

Theorem {595}.  CORRECTNESS-OF-ARRAY-QEMPTYP (rewrite):
(IMPLIES (ARRAY-QUEUEP QUEUE)
         (EQUAL (ARRAY-QEMPTYP QUEUE)
                (QEMPTYP (MAPUP-QUEUE QUEUE))))

Theorem {596}.  CORRECTNESS-OF-ARRAY-QFULLP:
(IMPLIES (AND (ARRAY-QUEUEP QUEUE)
              (EQUAL MAX
                     (GETNTH (QMAXLENGTH-FIELD) QUEUE)))
         (EQUAL (ARRAY-QFULLP QUEUE)
                (QFULLP (MAPUP-QUEUE QUEUE) MAX)))

Theorem {602}.  CORRECTNESS-OF-ARRAY-NONLAST (rewrite):
(IMPLIES (AND (ARRAY-QUEUEP QUEUE)
              (NOT (ARRAY-QEMPTYP QUEUE)))
         (EQUAL (MAPUP-QUEUE (ARRAY-NONLAST QUEUE))
                (NONLAST (MAPUP-QUEUE QUEUE))))

Theorem {603}.  CORRECTNESS-OF-ARRAY-QREPLACE (rewrite):
(IMPLIES (AND (ARRAY-QUEUEP QUEUE)
              (NOT (ARRAY-QEMPTYP QUEUE)))
         (EQUAL (MAPUP-QUEUE (ARRAY-QREPLACE ITEM QUEUE))
                (QREPLACE ITEM (MAPUP-QUEUE QUEUE))))
```

6.3 Using the Queue Correctness Theorems

We explain how the queue correctness theorems are used in the verification of KIT. The KIT source code contains subroutines for queue manipulations. The annotated text of the subroutine **ENQUEUE** is displayed below. See Section 4.2 for comments on how to read the source code.

```
ENQUEUE
;; Assume R2 contains item to enqueue
;;           R3 points to queue
;; this routine assumes queue not currently full
;; pseudo-code:
;;    store the item where ever the tail index points
;;    increment the current length
;;    increment the tail index (mod max-index)
(move  (1 r4) (1 r3))
(add   (1 r4) qarray-field)
(add   (1 r4) (3 r3 qtail-field))  ;; r4 has address of free slot
(move  (3 r4) (1 r2))              ;; store item
(incr  (3 r3 qcurrlength-field))   ;; increment current length
(incrm (3 r3 qtail-field) (3 r3 qmaxlength-field)) ;; increment tail
(return)
```

One might expect that we would state an entry and exit specification for **ENQUEUE** and prove a theorem which embodies its correctness. We have not chosen this

approach because of the ugly theorem which arises. Recall that we are verifying at the machine code level. Programs reside in a flat address space. The statement of a correctness theorem for **ENQUEUE** must include conditions such as "the program counter contains the address of the first instruction in **ENQUEUE**". Due to the flat address space, the statement of the theorem would change whenever we make a change to KIT which moves the starting address of **ENQUEUE**.

Our approach is to ignore subroutines. When we prove a lemma which traces a path through a call to **ENQUEUE**, we recognize an expression which matches the definition of **ARRAY-ENQ** and arrange for the rewriter to fold the expression up into a call to **ARRAY-ENQ**. The lemma **CONTRACT-ARRAY-ENQ** accomplishes this. Immediately upon encountering a sequence of **PUTNTH**s which matches the form of the definition of **ARRAY-ENQ**, the rewriter replaces the expression by an **ARRAY-ENQ** form.

```
Theorem {2079}.  CONTRACT-ARRAY-ENQ (rewrite):
(IMPLIES
 (EQUAL MAXLENGTH (GETNTH (QMAXLENGTH-FIELD) QUEUE))
 (EQUAL (PUTNTH (INCR-MOD (GETNTH (QTAIL-FIELD) QUEUE) MAXLENGTH)
                (QTAIL-FIELD)
                (PUTNTH (ADD1 (GETNTH (QCURRLENGTH-FIELD) QUEUE))
                        (QCURRLENGTH-FIELD)
                        (PUTNTH ITEM
                                (PLUS (QARRAY-FIELD)
                                      (GETNTH (QTAIL-FIELD) QUEUE))
                                QUEUE)))
        (ARRAY-ENQ ITEM QUEUE)))
```

The functions like **OS-CLOCK-INTERRUPT-HANDLER** which express the state of the machine at the end of an **OS** interrupt handler have already made a small step across the gap from the operating system layer to the abstract kernel layer. Array manipulation expressions are bundled up into calls to the **ARRAY-QUEUE** primitives, which have been independently verified to implement abstract queues.

Chapter 7

Conclusion

7.1 Related Work

We review three areas of related work: the program verification techniques upon which our work is based, previous attempts to verify operating systems, and microprogram verification.

7.1.1 Specification and Proof Methods

Our approach to the specification and verification of KIT derives from well known earlier work. The *implements* relation established by an interpreter equivalence theorem is an instance of Milner's *weak simulation* relation [Milner]. Hoare's approach to proving the correctness of data representations [Hoare], similar to Milner's work, is also a precursor. The application of Hoare's method can be most clearly seen in our treatment of queues.

Several attempts to verify operating systems cite the work of Milner, Hoare and others who have suggested similar approaches to verification. The methodology for designing operating system software proposed by Robinson and his co-workers [Robinson] calls for a sequence of abstract machines, each related by an *implements* relation. Kemmerer [Kemmerer] acknowledges a debt to Milner and Hoare in applying the Alphard methodology [Alphard] to the verification of a portion of the security kernel of UCLA Secure Unix. Rushby [Rushbyc] described an approach to kernel verification similar to ours. Hunt [Hunt85] proved an interpreter equivalence theorem to establish the correctness of a microprocessor.

7.1.2 Operating System Verification

Two areas predominate in operating system verification: verification of parallel processes, and verification of security properties.

The correctness of parallel programs is a large area we do not attempt to review exhaustively. The area of parallelism is primarily concerned with proving safety and liveness properties of sets of processes under various models of process communication. Above the kernel level, an operating system can be viewed as a set of cooperating parallel processes. So, techniques for verifying parallel processes can be applied to operating system verification above the kernel level. Our work logically supports this work. The purpose of our work is to verify a kernel implementation. We don't reason about the correctness of a particular set of concurrent processes, but prove that any set of processes which can be implemented on KIT is implemented without errors introduced by KIT.

We mention a number of efforts in operating system verification whose main contributions are in techniques for verifying parallel programs. The seminal work in this area is the "THE"-multiprogramming system [Dijkstra68] in which process synchronization via semaphores is implemented at the lowest layer. This work reveals to what advantage an operating system can be designed as a system of communicating sequential processes.

Saxena [Saxena] considers low level issues of processor and memory sharing in a multiprogrammed operating system. The design of a scheduler and memory manager, synchronized via monitors, is verified. A design methodology involving hierarchical decomposition and structured programming is discussed.

Flon's dissertation [Flon77] treats two subjects related to the correctness of operating systems. First, a methodology for the design, implementation and verification of operating systems is discussed. This methodology employs data abstraction to implement modular programs. A simple process dispatcher is specified, implemented and verified. Second, the problem of the total correctness of parallel programs is considered.

Karp [Karp83] proposes an extension of Pascal to include a method of process communication called a *module*, similar to a Simula Class. Concurrent systems expressed in this language can be demonstrated to be *failure free*, which is a notion of non-termination. The application of this communication model to operating systems is demonstrated.

Security is the other major area in operating system verification. In the early seventies the notion became current that a security policy should be implemented in the nucleus of an operating system, a *security kernel*. A number of efforts attempted to design, implement and verify a security kernel. A security policy given by Bell and LaPadula [BellLapadula] was the first attempt to formalize a specification for a security kernel. Alternative formulations of security were given by Feiertag, Levitt and Robinson [FeiertagModel], and by Popek and Farber [Popek].

The goals of each security kernel project were similar in outline: design a security kernel, prove that the design satisfies a formally described security policy, implement the kernel, and prove the implementation correct. Some projects were intended to complete only an initial portion of this sequence of goals. The goals were met with varying success.

Neumann and co-workers designed a provably secure operating system (PSOS) [PSOS, PSOSFinalReport] based on a capability mechanism. Parts of the design proof were sketched. An implementation was not completed. The main result of the project was a hierarchical methodology for operating system design [Robinson].

A group at Ford Aerospace designed a kernelized secure operating system (KSOS) [KSOS, KSOSDevelopment] intended to provide a secure operating system with an interface compatible with UNIX. The security policy for KSOS was approximately the Bell and LaPadula model. Information flow theorems at the design level were checked on the Boyer-Moore theorem prover. An implementation was written in MODULA, but code proofs were not anticipated and not done. The KSOS project benefited from the design methodology developed for PSOS.

The UCLA Secure Unix project [UCLASecureUnix, Walker] had a goal similar to the KSOS project: a Unix system built on a security kernel. The top level security policy was based on Popek and Farber's security model. The policy was enforced by a policy manager process outside the kernel. The kernel is responsible for manipulating processes and capabilities as allowed by the policy manager. This security kernel is more completely verified than the others. Kemmerer's dissertation [Kemmerer] reports on a design proof for the security kernel. The kernel was implemented in an extended version of Pascal, and some code level proofs were completed in the XIVUS verification system [XIVUS].

Other security kernels are reported in the literature, including the KVM/370 project [KVM370], and SCOMP [SCOMP]. The Secure Ada Target (SAT, now called LOCK) [SAT] is an ongoing project at Honeywell. Landwehr [Landwehr] gives a useful summary of the state of the art circa 1983.

Rushby criticizes the kernel approach to system security [Rushbya]. We do not repeat his argument, but point out that the alternative approach to security which he proposes results in a mandate for the type of verification carried out for KIT: a proof of the isolation of processes implemented in a shared environment.

The relationship between our work and that reported in the literature can be summarized as follows. There are two main threads in operating system verification: verification of parallel processes, and verification of security. The work in parallel processes lies inherently above the level of verification reported for KIT. The work in security reaches in principle down to the implementation level of KIT, but no one has previously reached that level.

7.1.3 Microprogram Verification

We mention the subject of microprogram verification to indicate its relation to our work. The goal of microprogram verification is to prove the correctness of an implementation of a machine architecture. Our work is based on a specification for a machine architecture, so our work lies logically just above the level of microprogram verification. Conceptually, the two areas can be merged. By targeting a verified kernel to a verified architecture we can combine the two levels of verification to span a much larger implementation gap.

The techniques of microprogram verification are similar to ours. The correctness theorem is stated as a machine simulation relation - an architecture level at the abstract end, and a register-transfer level at the low end. Paths through microcode are traced to relate a series of low steps to a high step. See [Hunt85], [Joyner], [SDVS] for examples of this work.

7.2 Comments and Summary

7.2.1 The Size of the KIT Project

The KIT project was conceived as an attempt to prove task isolation in a kernel written for a very simple von Neumann machine. We placed the following requirements on the problem. We felt that the combination of these constraints would force us to confront issues not before treated in operating system verification.

- Tasks must be able to communicate by some means. Therefore, task isolation really means limited task communication.
- The target machine's architecture must contain a very simple protection mechanism. We did not want the architecture to be so powerful that the entire problem would be solved at that level.
- The target machine must permit communication with asynchronous devices. Therefore, the operating system must field interrupts.
- All code must be verified. This meant that we would verify machine code.

The first three months were spent in an attempt to define the problem in such a way that the only property which required verification was task isolation. We felt, for

instance, that the verification of a particular scheduling algorithm was beyond the scope of this work. We also felt that the verification of various data structure implementations, particularly queues, were not of primary concern. We failed in our attempt to separate concerns. The reason for this was our requirement for an extremely simple target machine. With such a simple machine, we could not isolate the aspects we hoped not to verify. Therefore, everything necessary to create a completely operational, but simple, kernel was included.

A year passed. In that time we defined a prototype task, abstract kernel and target machine. We proved that the abstract kernel implemented a task. We proved several kernel routines including the clock interrupt handler and error handler. In doing so we learned the overall structure of the kernel proof. We learned our technique for making the theorem prover symbolically execute machine code. We went through several revisions of our theory of arrays. This was our first experience in using the Boyer-Moore theorem prover for a proof deeper than **ASSOCIATIVITY-OF-APPEND**, although we were already familiar with the Boyer-Moore logic.

At this point, with our support theories well in hand, we started the project almost from scratch. We defined the target machine to be simpler than the prototype had been. We revised our specifications for the abstract kernel. We wrote the complete kernel and proved its correctness. This took three months.

The size of the script is extremely large. We attribute this primarily to the inherent complexity of the problem. There is simply a large gap to span from a target machine as simple as **TM** to the level of our abstract kernel. The bulk of our script is devoted to three areas.

- The trace lemmas which result in the definition of the operating system layer.
- The proof of the operating system layer invariant.
- The proof of the correctness of the operating system layer - i.e. that the operating system layer implements the abstract kernel.

These are large because an enormous number of cases must be considered. We must

prove that each of thirty-eight paths through the kernel correctly manipulates each of ten abstract kernel fields, most of which are structured objects. The trace lemmas were the most difficult to check. The theorem prover required much help by way of rewrite lemmas to symbolically execute the address computations which occurred in each path. We found, though, that while the initial lemmas in each of the three proofs above were slow going, later proofs became progressively easier. Toward the end we generated the lemmas we needed by merely editing previous lemmas.

To understand the size of the script, one must also consider the starting point: the elementary theory of numbers and lists built into the Boyer-Moore theorem prover. Much groundwork was required in terms of facts about arithmetic, sets and arrays. The script contains a complete target machine definition and operating system specification. There is much more in the script than the KIT code and its proof.

The verification revealed bugs in the operating system code. Simple bugs, like naming an incorrect register, or using the wrong address mode, were revealed at the time a tracing lemma was proved. During tracing it became obvious when a data structure was addressed incorrectly. More difficult bugs were revealed during the proof that each KIT routine implements the corresponding abstract kernel operation. The most insidious bug revealed at this stage was one in which the state of the current task was not accurately restored after processing an I/O interrupt. The bug would have caused a supervisor call request to be ignored if an I/O interrupt occurred immediately after the request, but before the request had been handled. Such time-dependent errors are difficult to find by testing.

KIT is so small that it is likely that a group of competent programmers could produce in a short time a highly reliable version using traditional coding and testing techniques. Without the goal of mechanical verification, it is unlikely that the specification for KIT would be stated as explicitly as we have done. Therefore, it is questionable whether all the issues which we encountered during our proof would be considered by traditional means. In particular, the proof that the target machine permits

the implementation of isolated address spaces would likely have to be assumed. If a programming team got so far as to state a specification in as much detail as our abstract kernel, it is unlikely that a hand proof of KIT with respect to this specification would be convincing. The proof is so large that a mechanical check is virtually a necessity in making sure that all cases have been considered.

7.2.2 The Significance of the KIT Project

The purpose of KIT is to provide verified task isolation. That is, tasks can communicate only in specified ways. As a result, a verified set of communicating processes will run as specified on KIT provided there are no hardware errors. KIT is guaranteed not to introduce implementation bugs, since all code is verified.

A number of significant results are required to establish the main theorem.

- The termination of kernel routines.
- The correctness of the address space abstraction, i.e., that an address space can be viewed as an independent machine.
- The isolation of the operating system from tasks on the target machine.
- The inability of a task to enter supervisor mode.

Therefore, the verification of KIT checks important security properties. We have proved task isolation, the protection of the operating system from tasks, and the inability of tasks to enter supervisor mode. Our small system is tamper proof. These results are fundamental to computer security but have received scant attention in formal verification. Previous attempts to verify security have been concerned with models of security in which data and processes are tagged with security levels. The issues involved in correctly implementing multiple processes on shared resources have been largely ignored.

The proof is accomplished by establishing a machine simulation theorem which relates KIT to a definition of a process which appears to be running on its own machine. KIT is shown to implement a fixed number of conceptually distributed communicating processes. The specification machine is so abstract that the proof of its properties is quite

tractable. An example of a property which is trivial to establish at this level is the protection of a process's private state. We have not stated and proved other properties, but clearly it is preferable to do so at the high end than at the low end. Because the *implements* relation is proved, properties established at the high end hold (under some state space mapping) at the low end.

There is a technical advantage in pushing operational specifications to as abstract a level as possible in the Boyer-Moore logic, and using machine simulation theorems to establish correctness. The advantage is that the Boyer-Moore theorem prover's definitional principle gives a proof of the unique existence of every function defined, and therefore a proof of consistency of the specification. If our method had been to prove a set of properties stated directly about the implementation of KIT, then not only would their statement have been difficult, but the consistency issue would have been confronted.

Nearly all the difficulties in our proof occurred in establishing the *implements* relation between the operating system running on the target machine (the **os** layer, see Figure 5-1) and the abstract kernel. These difficulties were largely due to issues unrelated to task isolation: the verification of queues, tracing paths through the operating system code. We have found no good solutions to the problem of verifying machine code. Our method is shown to work for a small example, but whether it is tractable for a large system is an open question.

What we can learn from the exercise is the structure which the proof of a kernel may take: a machine simulation theorem between an abstract kernel and the kernel implementation. The abstract kernel makes much simpler a number of issues which are quite complex at the kernel implementation level: the termination of kernel operations, low level representation of data structures, isolation of processes. Making sure that the interrupt structure of the abstract kernel is identical to the interrupt structure of the target machine makes possible an inductive proof of the machine simulation theorem. Even if, in a larger system, a mechanical proof of a kernel implementation is unfeasible, the

existence of a specification at the level of KIT's abstract kernel gives a good guide for hand verification.

The exercise of verifying KIT also reveals some necessary properties at the architecture level which make the proof possible. We have formalized the notion of an address space for our simple target machine, and proved the correctness and protection theorems which make it possible to view an address space as an isolated private machine. In a future in which hardware is formally specified and verified, such theorems can be checked early about a hardware design.

7.2.3 Future Work

This work can be carried forward. More complex phenomena in several areas may be considered. At the top end, more sophisticated methods of inter-task communication may be specified, e.g. shared segments and files. An obvious deficiency in KIT from the point of view of general purpose operating systems is the absence of dynamic process and channel creation. These issues should get attention if we hope to verify usable general purpose systems. Fixed systems like KIT, though, do have their applications, such as communications processing. Due to the difficulty of verifying large amounts of machine code, these issues may not be tractable until we find a way to verify a high-level language version of the kernel.

At the low end, more complex architectures can be considered to great advantage. We restricted this work to an extremely simple hardware protection mechanism. A more sophisticated protection mechanism can make the isolation proof much easier. Of great interest, and in a slightly different vein, are the real-time properties of a system. Although we have not proved such properties, it is possible to consider proofs of response time to external events. It would be worthwhile to relax the property of the non-interruptibility of the supervisor for such proofs. Considering such low-level phenomena at the hardware/software boundary may have some immediate impact since, if our experience with KIT is any indication, proofs at this level tend to be relatively short.

KIT's message passing mechanism is a subset of that specified for the programming language Gypsy [Gypsy205, GypsyConcurrency]. Given the right compiler, it is possible to think of KIT as a verified run-time environment for a subset of Gypsy. Accomplishing this is another goal for the future.

Appendix A.

The Boyer-Moore Logic and its Theorem Prover[1]

In [acl] we describe a quantifier free first-order logic and a large and complicated computer program that proves theorems in that logic. The major application of the logic and theorem prover is the formal verification of properties of computer programs, algorithms, system designs, etc. In this section we describe the logic and the theorem prover.

A.1 The Logic

A complete and precise definition of the logic can be found in Chapter III of [ACL] together with the minor revisions detailed in section 3.1 of [META].

We use the prefix syntax of Pure Lisp to write down terms. For example, we write **(PLUS I J)** where others might write **PLUS(I,J)** or **I+J**.

The logic is first-order, quantifier free, and constructive. It is formally defined as an extension of propositional calculus with variables, function symbols, and the equality relation. We add axioms defining the following:

- the Boolean objects **(TRUE)** and **(FALSE)**, abbreviated **T** and **F**;
- The if-then-else function, **IF**, with the property that **(IF x y z)** is **z** if **x** is **F** and **y** otherwise;
- the Boolean "connector functions" **AND**, **OR**, **NOT**, and **IMPLIES**; for example, **(NOT p)** is **T** if **p** is **F** and **F** otherwise;
- the equality function **EQUAL**, with the property that **(EQUAL x y)** is **T** or **F** according to whether **x** is **y**;

[1]Written by Boyer and Moore. Taken with permission from [quant].

- inductively constructed objects, including:
  - Natural Numbers. Natural numbers are built from the constant **(ZERO)** by successive applications of the constructor function **ADD1**. The function **NUMBERP** recognizes natural numbers, e.g., is **T** or **F** according to whether its argument is a natural number or not. The function **SUB1** returns the predecessor of a non-**0** natural number.
  - Ordered Pairs. Given two arbitrary objects, the function **CONS** returns an ordered pair containing them. The function **LISTP** recognizes such pairs. The functions **CAR** and **CDR** return the two components of such a pair.
  - Literal Atoms. Given an arbitrary object, the function **PACK** constructs an atomic symbol with the given object as its "print name." **LITATOM** recognizes such objects and **UNPACK** returns the print name.
- We call each of the classes above a "shell." **T** and **F** are each considered the elements of two singleton shells. Axioms insure that all shell classes are disjoint;
- the definitions of several useful functions, including:
  - **LESSP** which, when applied to two natural numbers, returns **T** or **F** according to whether the first is smaller than the second;
  - **LEX2**, which, when applied to two pairs of naturals, returns **T** or **F** according as whether the first is lexicographically smaller than the second; and
  - **COUNT** which, when applied to an inductively constructed object, returns its "size;" for example, the **COUNT** of an ordered pair is one greater than the sum of the **COUNT**s of the components.

The logic provides a principle under which the user can extend it by the addition of new shells. By instantiating a set of axiom schemas the user can obtain a set of axioms describing a new class of inductively constructed **n**-tuples with type-restrictions on each component. For each shell there is a recognizer (e.g., **LISTP** for the ordered pair shell), a constructor (e.g., **CONS**), an optional empty object (e.g., there is none for the ordered pairs but **(ZERO)** is the empty natural number), and **n** accessors (e.g., **CAR** and **CDR**).

The logic provides a principle of recursive definition under which new function symbols may be introduced. Consider the definition of the list concatenation function:

```
Definition.
(APPEND X Y)
   =
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y).
```

The equations submitted as definitions are accepted as new axioms under certain conditions that guarantee that one and only one function satisfies the equation. One of the conditions is that certain derived formulas be theorems. Intuitively, these formulas insure that the recursion "terminates" by exhibiting a "measure" of the arguments that decreases, in a well-founded sense, in each recursion. A suitable derived formula for **APPEND** is:

```
(IMPLIES (LISTP X)
         (LESSP (COUNT (CDR X))
                (COUNT X))).
```

However, in general the user of the logic is permitted to choose an arbitrary measure function (**COUNT** was chosen above) and one of several relations (**LESSP** above).

The rules of inference of the logic, in addition to those of propositional calculus and equality, include mathematical induction. The formulation of the induction principle is similar to that of the definitional principle. To justify an induction schema it is necessary to prove certain theorems that establish that, under a given measure, the inductive hypotheses are about "smaller" objects than the conclusion.

Using induction it is possible to prove such theorems as the associativity of **APPEND**:

```
Theorem.
(EQUAL (APPEND (APPEND A B) C)
       (APPEND A (APPEND B C))).
```

A.2 The Mechanization of the Logic

The theorem prover for the logic, as it stood in 1979, is described completely in [acl]. Many improvements have been added since. In [Meta] we describe a "metafunction" facility which permits the user to define new proof procedures in the logic, prove them correct mechanically, and have them used efficiently in subsequent proof attempts. During the period 1980-1985 a linear arithmetic decision procedure was

integrated into the rule-driven simplifier. The problems of integrating a decision procedure into a heuristic theorem prover for a richer theory are discussed in [Linear]. The theorem prover is briefly sketched here.

The theorem prover is a computer program that takes as input a term in the logic and repeatedly transforms it in an effort to reduce it to non-**F**. The theorem prover employs eight basic transformations:

- decision procedures for propositional calculus, equality, and linear arithmetic;
- term rewriting based on axioms, definitions and previously proved lemmas;
- application of verified user-supplied simplifiers called "metafunctions;"
- renaming of variables to eliminate "destructive" functions in favor of "constructive" ones;
- heuristic use of equality hypotheses;
- generalization by the replacement of terms by type-restricted variables;
- elimination of apparently irrelevant hypotheses; and
- mathematical induction.

The theorem prover contains many heuristics to control the orchestration of these basic techniques.

In a shallow sense, the theorem prover is fully automatic: the system accepts no advice or directives from the user once a proof attempt has started. The only way the user can alter the behavior of the system during a proof attempt is to abort the proof attempt. However, in a deeper sense, the theorem prover is interactive: the system's behavior is influenced by the data base of lemmas which have already been formulated by the user and proved by the system. Each conjecture, once proved, is converted into one or more "rules" which guide the theorem prover's actions in subsequent proof attempts.

A data base is thus more than a logical theory: it is a set of rules for proving theorems in the given theory. The user leads the theorem prover to "difficult" proofs by "programming" its rule base. Given a goal theorem, the user generally discovers a proof

himself, identifies the key steps in the proof, and then formulates them as lemmas, paying particular attention to their interpretation as rules.

The key role of the user in our system is guiding the theorem prover to proofs by the strategic selection of the sequence of theorems to prove and the proper formulation of those theorems. Successful users of the system must know how to prove theorems in the logic and must understand how the theorem prover interprets them as rules.

Appendix B.

Index of Events in this Volume

AK-BLOCK-INPUT 27
AK-BLOCK-OUTPUT 29
AK-BLOCK-RECEIVE 26
AK-BLOCK-SEND 25
AK-CHANNELS 92
AK-CLOCK-INTERRUPT-HANDLER 23
AK-DISPATCHER 23
AK-ERROR-HANDLER 24
AK-EXECUTE-INPUT 27
AK-EXECUTE-INPUT-FROM-BUFFER 28
AK-EXECUTE-OUTPUT 28
AK-EXECUTE-OUTPUT-TO-BUFFER 29
AK-EXECUTE-RECEIVE 26
AK-EXECUTE-RECEIVE-FROM-BUFFER 27
AK-EXECUTE-SEND 25
AK-EXECUTE-SEND-TO-BUFFER 26
AK-FETCH-EXECUTE 22
AK-IMPLEMENTS-ACTIVE-TASK-STEP 97
AK-IMPLEMENTS-NON-ACTIVE-TASK-STEP 97
AK-IMPLEMENTS-PARALLEL-TASKS 10, 92, 94
AK-INPUT-INTERRUPT-HANDLER 29
AK-OUTPUT-INTERRUPT-HANDLER 31
AK-PRIVATE-STEP 22
AK-PROCESSOR 21
AK-RUNNING-INPUT-INTERRUPT-HANDLER 30
AK-RUNNING-OUTPUT-INTERRUPT-HANDLER 32
AK-STEP 22
AK-SVC-HANDLER 25
AK-TASKID 22
AK-TASKIDLUB 20
AK-WAITING-INPUT-INTERRUPT-HANDLER 30
AK-WAITING-OUTPUT-INTERRUPT-HANDLER 32
ARRAY-DEQ 102
ARRAY-ENQ 102
ARRAY-NONLAST 102
ARRAY-QEMPTYP 102
ARRAY-QFIRST 102
ARRAY-QFULLP 102
ARRAY-QINDEX-RELATION 101
ARRAY-QREPLACE 102
ARRAY-QUEUEP 101
CONTRACT-ARRAY-ENQ 105
CONTROL-ORACLE 93
CONTROL-ORACLE-STEP 94
CORRECTNESS-OF-ARRAY-DEQ 103
CORRECTNESS-OF-ARRAY-ENQ 103
CORRECTNESS-OF-ARRAY-NONLAST 104
CORRECTNESS-OF-ARRAY-QEMPTYP 104

CORRECTNESS-OF-ARRAY-QFIRST 104
CORRECTNESS-OF-ARRAY-QFULLP 104
CORRECTNESS-OF-ARRAY-QREPLACE 104
CORRECTNESS-OF-CLOCK-INTERRUPT-HANDLER 89
CORRECTNESS-OF-OPERATING-SYSTEM 10, 98
DECR-MOD 102
DELTA 101
DELTA-EQUALS-LENGTH-DELTA-SEGMENT 103
DELTA-SEGMENT 103
DEQ 18, 99
ENQ 18, 99
FINITE-NUMBER-LISTP 20
FINITE-NUMBERP 20
FOO 5
GETNTH 17
GOOD-ADDRESS-SPACE 87
GOOD-AK 20
GOOD-OS 75
GOOD-TASK 14
GOOD-TM 36
INCR-MOD 102
MAPUP-ADDRESS-SPACE 86
MAPUP-BASE 86
MAPUP-CC 86
MAPUP-CPU 87
MAPUP-CURRENT-TASK-TM-FETCH-EXECUTE 90
MAPUP-ERROR 86
MAPUP-LIMIT 86
MAPUP-OS 84
MAPUP-OS-IBUFFERS 85
MAPUP-OS-MBUFFERS 85
MAPUP-OS-OBUFFERS 85
MAPUP-OS-TASKS 85
MAPUP-QUEUE 103
MAPUP-QUEUE-LIST 84
MAPUP-REGS 86
MAPUP-SVCFLAG 86
MAPUP-SVCID 86
MAPUP-TASK 85
MAPUP-TASK-SEPARATION 91
MAPUP-TASKS 85
NONLAST 99
OS-CLOCK-INTERRUPT-HANDLER 79
OS-CODE 77
OS-CODE-ADDRESS 77
OS-CODE-LENGTH 78
OS-IMPLEMENTS-AK 84
OS-IMPLEMENTS-PARALLEL-TASKS 10, 98
OS-MACHINE-CODE 78
OS-ORACLE 82
OS-ORACLE-STEP 83
OS-PROCESSOR 76
OS-STEP 77
OS-STEP-IMPLEMENTS-AK-STEP 87
PROJECT 92
PUTNTH 17
QARRAY-FIELD 100
QCURRLENGTH-FIELD 100
QEMPTYP 18, 99
QFIRST 17, 99
QFULLP 18, 99

QHEAD-FIELD 100
QMAXLENGTH-FIELD 100
QREPLACE 18, 99
QTAIL-FIELD 100
REAL-ADDR 40
REAL-ADDR-NUM 40
REAL-ADDR-SOURCE 40
TABLE 76
TASK-ACTIVEP 15
TASK-EXECUTE-INPUT 17
TASK-EXECUTE-OUTPUT 17
TASK-EXECUTE-RECEIVE 16
TASK-EXECUTE-SEND 16
TASK-FETCH-EXECUTE 18
TASK-IBUFFERS 14
TASK-MBUFFERS 14
TASK-OBUFFERS 14
TASK-PRIVATE-STEP 15
TASK-PROCESSOR 15
TASK-STEP 15
TASK-UPDATE-CHANNELS 15
TIMED-TM-PROCESSOR 81
TIMED-TM-STEP 82
TM-CC-VALUE 43
TM-EXECUTE-ADD 43
TM-EXECUTE-CLOCK-INTERRUPT 41
TM-FETCH 39
TM-FETCH-EXECUTE 42
TM-FETCH-EXECUTE-COMMUTES-WITH-MAPUP-ADDRESS-SPACE 91
TM-FETCH-EXECUTE-MAPUP-ADDRESS-SPACE-SEPARATION 92
TM-FETCH-FROM-MEMORY 40
TM-FETCH-FROM-REGMEM 40
TM-FETCH-NEW-PC-ON-INTERRUPT 41
TM-IMPLEMENTS-OS 83
TM-IMPLEMENTS-TIMED-TM 83
TM-PORT-LENGTH 36
TM-POST-INPUT-INTERRUPT 38
TM-POST-INTERRUPT 37
TM-POST-OUTPUT-INTERRUPT 38
TM-PROCESSOR 37
TM-REGISTER-SAVE-AREA-ADDR 41
TM-SET-CC 39
TM-STEP 38
TM-STORE 40
TM-STORE-IN-MEMORY 40
TM-STORE-IN-REGMEM 40
TM-STORE-OLD-PSW-ON-INTERRUPT 41
TM-WORDLUB 36
TM-WORDSIZE 36
TRACE-CLOCK-INTERRUPT-HANDLER 80

References

[Bach 86] M.J. Bach.
The Design of the UNIX Operating System.
Prentice-Hall, Englewood Cliffs, N.J., 1986.

[Bell 71] C.G. Bell, A. Newell.
Computer Structures: Readings and Examples.
McGraw-Hill, New York, 1971.

[Bell 75] D.E. Bell, L.J. LaPadula.
Secure Computer Systems: Unified Exposition and Multics Interpretation.
Technical Report MTR-2997, The Mitre Corporation, July, 1975.

[Berson 79] T.A. Berson, G.L. Barksdale, Jr.
KSOS - Development Methodology for a Secure Operating System.
In *AFIPS Conference Proceedings*, pages 365-371. 1979.

[Boebert 85] W.E. Boebert, W.D. Young, R.Y. Kain, S.A. Hansohn.
Secure Ada Target: Issues, System, Design, and Verification.
In *Proceedings of the Symposium on Security and Privacy*, pages 176-183. 1985.

[Boyer 79] R. S. Boyer, J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer 81] R.S. Boyer, J S. Moore.
Metafunctions: Proving Them Correct and Using them Efficiently as New Proof Procedures.
The Correctness Problem in Computer Science.
Academic Press, London, 1981.

[Boyer 85] R.S. Boyer, J S. Moore.
Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic.
Technical Report ICSCA-CMP-44, Institute for Computing Science, University of Texas at Austin, January, 1985.

[Boyer 87] R.S. Boyer, J S. Moore.
The Addition of Bounded Quantification and Partial Functions to A Computational Logic and its Theorem Prover.
Technical Report ICSCA-CMP-52, Institute for Computing Science, University of Texas at Austin, January, 1987.

[BrinchHansen 70]
P. Brinch Hansen.
The Nucleus of a Multiprogramming System.
CACM 13(4):238-241,250, April, 1970.

[Dijkstra 68] E.W. Dijkstra.
The Structure of the "THE"-Multiprogramming System.
CACM 11(5):341-346, May, 1968.

[Feiertag 77] R.J. Feiertag, K.N. Levitt, L. Robinson.
Proving Multilevel Security of a System Design.
In *Proceedings 6th ACM Symposium on Operating System Principles*, pages 57-65. 1977.

[Feiertag 79] R.J. Feiertag, P.G. Neumann.
The Foundations of a Provably Secure Operating System (PSOS).
In *AFIPS Conference Proceedings*, pages 329-334. 1979.

[Flon 77] L. Flon.
On the Design and Verification of Operating Systems.
PhD thesis, Carnegie-Mellon University, 1977.

[Fraim 83] L. Fraim.
Scomp: A Solution to the Multilevel Security Problem.
Computer 16(7):26-34, July, 1983.

[Gold 79] B.D. Gold, R.R.Linde, R.J. Peeler, M. Schaefer, J.F. Scheid, P.D. Ward.
A Security Retrofit of VM/370.
In *AFIPS Conference Proceedings*, pages 335-344. 1979.

[Good 75] D.I. Good, R.L. London, W.W. Bledsoe.
An Interactive Program Verification System.
IEEE Transactions on Software Engineering 1(1):59-67, March, 1975.

[Good 79] D.I. Good, R.M. Cohen, J. Keeton-Williams.
Principles of Proving Concurrent Programs in Gypsy.
Technical Report ICSCA-CMP-15, Institute for Computing Science, University of Texas at Austin, January, 1979.

[Good 86] D.I. Good, R.L. Akers, L.M. Smith.
Report on Gypsy 2.05.
Technical Report 1, Computational Logic, Inc, October, 1986.

[Hoare 72] C.A.R. Hoare.
Proof of Correctness of Data Representations.
Acta Informatica 1:271-281, 1972.

[Hunt 85] W.A. Hunt, Jr.
FM8501: A Verified Microprocessor.
Technical Report 47, Institute for Computing Science, University of Texas at Austin, December, 1985.

[Joyner 76] W.H. Joyner, G.B. Leeman, W.C. Carter.
Automated Verification of Microprograms.
Technical Report, IBM Thomas J. Watson Research Center, April, 1976.

[Karp 83] R.A. Karp.
Proving Operating Systems Correct.
UMI Research Press, Ann Arbor, Michigan, 1983.

[Kemmerer 82] R.A. Kemmerer.
Formal Verification of an Operating System Security Kernel.
UMI Research Press, Ann Arbor, Michigan, 1982.

[Landwehr 83] C.E. Landwehr.
The Best Available Technologies for Computer Security.
Computer 16(7):86-100, July, 1983.

[Marcus 84] L. Marcus, S.D. Crocker, J.R. Landauer.
SDVS: A System for Verifying Microcode Correctness.
In *The Seventh Annual Microprogramming Workshop*, pages 246-255. 1984.

[McCauley 79] E.J. McCauley, P.J. Drongowski.
KSOS - The Design of a Secure Operating System.
In *AFIPS Conference Proceedings*, pages 345-353. 1979.

[Milner 71] R. Milner.
An Algebraic Definition of Simulation Between Programs.
Technical Report AIM-142, Stanford AI Project, February, 1971.

[Neumann 77] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, L. Robinson.
A Provably Secure Operating System: The System, Its Applications, and Proofs.
Technical Report, SRI, February, 1977.

[Popek 78] G.J. Popek, D.A. Farber.
A Model for Verification of Data Security in Operating Systems.
CACM 21(9):737-749, September, 1978.

[Popek 79] G.J. Popek, M. Kampe, C.S. Kline, A. Stoughton, M. Urban, E. Walton.
UCLA Secure Unix.
In *AFIPS Conference Proceedings*, pages 355-364. 1979.

[Robinson 77] L. Robinson, K.N. Levitt, P.G. Neumann, A.R. Saxena.
A Formal Methodology for the Design of Operating System Software.
Current Trends in Programming Methodology, Volume I: Software Specification and Design.
Prentice-Hall, Englewood Cliffs, N.J., 1977.

[Rushby 81a] J. Rushby.
Proof of Separability: A Verification Technique for a Class of Security Kernels.
Technical Report SSM/8, Computing Laboratory, University of Newcastle upon Tyne, May, 1981.

[Rushby 81b] J. Rushby.
Specification and Design of Secure Systems.
Technical Report SSM/6, Computing Laboratory, University of Newcastle upon Tyne, March, 1981.

[Saxena 76] A.R. Saxena.
A Verified Specification of a Hierarchical Operating System.
PhD thesis, Stanford University, 1976.

[Walker 80] B.J. Walker, R.A. Kemmerer, G.J. Popek.
Specification and Verification of the UCLA Unix Security Kernel.
CACM 23(2):118-131, February, 1980.

[Wulf 76] W.A. Wulf, R.L. London, M. Shaw.
Abstraction and Verification in Alphard: Introduction to Language and Methodology.
Technical Report ISI/RR-76-46, USC Information Sciences Institute, June, 1976.

VITA

William Richard Bevier, the son of M. Joan Hughes Bevier and Richard Herman Bevier, was born in Geneva, Illinois on July 15, 1950. He entered Beloit College in Beloit, Wisconsin in 1968, completing a B.A. in Mathematics in 1972. He completed an M.S. in Computer Science at Northern Illinois University during the years 1974 to 1977, where he was employed the following three years as an instructor in Computer Science. He entered the The Graduate School of the University of Texas in 1980. In 1982 he married Susan Francis Geis of Downers Grove, Illinois. Their son Henry Richard Bevier was born in 1984, and their daughter Anne Elizabeth Bevier was born in 1987.

Permanent Address: 821 Harris Ave.
Austin, Texas 78705

This dissertation was typed by the author.

TABLE OF CONTENTS

Acknowledgements iii

Abstract iv

Table of Contents v

Chapter 1. Introduction 1

1.1. The Thesis 1
1.2. Process Isolation 3
1.3. A Characterization of this Work 4
1.4. Plan of Dissertation 5
1.5. The Boyer-Moore Logic and its Proof Checker 5

Chapter 2. Defining Finite State Machines with Recursive Functions 7

2.1. Interpreters 7
2.2. Interpreter Equivalence Theorems 8
2.3. The KIT Proof Structure 10

Chapter 3. The Specification of KIT 13

3.1. The Task Layer 13
3.2. The Abstract Kernel Layer 20
3.2.1. The Clock Interrupt Handler 24
3.2.2. The Error Handler 25
3.2.3. The Supervisor Call Handler 26
3.2.3.1. Send 26
3.2.3.2. Receive 27
3.2.3.3. Input 28
3.2.3.4. Output 29
3.2.4. The Input Interrupt Handler 30
3.2.5. The Output Interrupt Handler 32

Chapter 4. The Implementation of KIT 35

4.1. The Target Machine 35
4.2. The Code 46
4.3. Flowcharts 60

Chapter 5. The Verification of KIT 74

5.1. The Operating System Layer 74
5.2. The Target Machine Implements the Operating System 82
5.3. The Operating System Implements the Abstract Kernel 85
5.4. The Abstract Kernel Implements Tasks 93
5.5. Composing the Interpreter Equivalence Theorems 99

Chapter 6. Queues 100

6.1. An Implementation of Queues 100
6.2. The Correctness of the Queue Implementation 104
6.3. Using the Queue Correctness Theorems 105

Chapter 7. Conclusion 107

7.1. Related Work 107
7.1.1. Specification and Proof Methods 107
7.1.2. Operating System Verification 108
7.1.3. Microprogram Verification 111
7.2. Comments and Summary 111
7.2.1. The Size of the KIT Project 111
7.2.2. The Significance of the KIT Project 114
7.2.3. Future Work 116

Appendix A. The Boyer-Moore Logic and its Theorem Prover[2] 118

A.1. The Logic 118
A.2. The Mechanization of the Logic 120

[2]Written by Boyer and Moore. Taken with permission from [Boyer 87].

Appendix B. Index of Events in this Volume 123
References 126
VITA 130

LIST OF TABLES

Table 4-1: PMS Description of TM 36
Table 4-2: The TM Clock Interrupt 42
Table 4-3: TM's Instruction Set 44
Table 4-4: Grammar for TM Assembler 48

LIST OF FIGURES

Figure 2-1: Interpreter Equivalence 9
Figure 2-2: KIT Proof Structure 11
Figure 3-1: Network 14
Figure 4-1: Layout of Kernel 47
Figure 5-1: Revised KIT Proof Structure 75
Figure 5-2: Traces of TM and OS 82
Figure 5-3: AK Implements Parallel Tasks 94
Figure 6-1: Delta 102