

VTOS: A Finite State Machine Model of OS and Formalized Verification of Its Microkernel

Hao Huang^{1*}, Fangmin Song¹, Zhenjiang Qian¹, Kangjie Li¹,
Jiang Chen¹, Shuiwang Guan¹ & Wei Huang²

¹*State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing China*

²*Department of Electrical and Computer Engineering, University of Toronto, Canada*

Abstract Operating system is a huge software with complicated functionality. The correctness of even single module cannot be proved through software test. Project seL4 has proved the correctness of each function of a microkernel operating system. The correctness of each single function of an operating system cannot prove its integrity and other security property. In this paper we proposed a OS model of finite state machine (OSMFSM). We design our VTOS(Verified Trusted Operating System), which is a microkernel operating system and aimed at being formally verified and trusted, according to this model. We define the state of the OSMFSM through the global objects in VTOS, and the transition function of the OSMFSM as a piecewise defined function through the functions in VTOS. Through establishing a domain for VTOS with the help of OSMFSM and another domain, which is isomorphic to the former, in Isabelle/HOL, we prove the correctness of some functions in the microkernel of VTOS formally. With the help of the concept of the invariability of the FSM, we propose a method of describing and proving the integrity of an OS.

Keywords OS FSM model, formal verification of OS, integrity of OS, micro-kernel OS, Isabelle/HOL

1 Introduction

Information security of computer systems is extremely important work in the information technology field. Advances in information security technology promote the process of informatization, however, in many areas it also unfavorably hampers the advance of informatization. Academia and industry has been conducted painstaking excellence in work. Notwithstanding the progress we made, the security of information systems is still full of uncertainty, and none of security products can guarantee full realization of the ideal security objectives. We believe that the security of the operating system (OS) is the foundation of computer system security.

Nevertheless, the enormous size and high complexity of the OS has made obstacles for developers to find defects and bugs in the progress of OS design and implementation, which unfortunately, could result in that the running OS be controlled by malicious programs.

An obvious better approach is to use rigorous mathematical methods to perform formal description and verification, *e.g.*, code analysis, model checking [?] and theorem proving. The work of design and verification with formal methods includes describing the behavior and semantics of the operations of the OS with strict mathematical methods, constructing OS semantic model, and verifying that the implementation meets predetermined functional requirements. This method is called formal verification.

*Corresponding author (email: hhuang@nju.edu.cn)

Formal verification has two approaches, *i.e.*, model checking and theorem proving. Model checking uses mathematical models to automatically reason and exhaustively check whether given specifications of system issues are satisfied. On the other hand, theorem proving generally uses interactive theorem provers to abstract and describe the system issues, and expresses the system's functionality and security in the form of mathematical formulae and theorems, and verifies the correctness with induction and deduction. It is worth mentioning that, comparing the two methods, more issues can be verified by theorem proving than by model checking.

Formal research work of operating system can be traced back to the seventies of the last century, and the early projects include UCLA Secure UNIX [?], PSOS [?], DTOS [?], EROS [?], VFiasco [?], Robin [?]. Verisoft XT [?] funded by the German BMBF aims to do pervasive formal verification of computer systems, *i.e.*, completely from the underlying hardware to the upper user program. seL4 microkernel [?,?] developed by NICTA originates from the third-generation microkernel of L4. Early in the design, the developers constructed the prototype in Haskell, and translated the Haskell implementation manually into high-performance C code. From abstract specifications to C code, the correctness of seL4 has been strictly verified in Isabelle/HOL [?]. The correctness verification [?,?] illustrates that the implementation of microkernel strictly follows the high-level abstract specification of behaviors of the kernel in the early design stage. Recently, seL4 researches how to verify the integrity of the system [?,?] and how to improve the performance [?,?]. The Flint project [?] led by Shao in Yale University studied the verification methods of program logics, and proposed a new language of logic verification (VeriML) [?,?], in which the logic system λ HOLind [?] adds ability of inductive definition of data types with capability of rich object expression and security features. In the mixed verification of various functional modules, Feng proposed an open logical framework OCAP [?], in which the verification logics for different modules are combined to form a complete verification system. They use the separation logic to verify the correctness of the software from the assembly level [?,?,?,?] and concurrent thread management [?]. The research group led by Chen in USTC contributes to the research of formal verification in the high-assurance software on constructing the infrastructure of large-scale system software with proof-carry code technology [?,?] . They study how to effectively integrate both the program formal verification and domain-specific language, to improve productivity of robust software and develop new methodology of software design.

In this paper, we use the formal method to describe and verify the correctness and security of the OS. We proposed a novel model of finite state machine for OS. OS is a complicated and huge software. Its functionality and security must be considered on the whole. A property possessed by a single function or single module can not convince people of the property of the whole OS. We describe the functionality and security properties with the help of this model, making it easier to figure out a logic formula for a functionality or a security property.

In Sec. 2 we define an operating system model using the concept of finite state machine. In Sec. 3 we define a domain for an the VTOS, and also define a domain in Isabelle/HOL which is isomorphic to the former. Based on this concept, we propose a method of proving the correctness and integrity of an OS in Isabelle/HOL. In Sec. 4 we explain the process of verifying the microkernel of VTOS in Isabelle/HOL. Finally, we conclude this paper in Sec. 5.

2 The model of finite state machine for OS

2.1 The essence of computing

A network server installed with operating system and network service software can continuously provide service after booting up. It results from the cooperative work between the hardware and software. Before we configure out an abstract operating system model, we have to analyze the function of each software and hardware component.

Modern computer consists of memory, arithmetic unit, and control unit. The memory includes registers in the CPU, caches, main memory, registers in the device controllers, exchange area in hard disk. The required instructions and data for each arithmetic unit and the computed results are all stored in the

memory. There is a small non-essential difference between these concept and normal concept in computer organization, because these new concept is more convenient for introducing our OS model.

The instructions run in the arithmetic units affect the data in the memory. The objects that can affect the data in the memory include arithmetic units in the CPU cores, and device controllers with DMA mechanism. All of the objects read or write the data in memory in parallel and we call these objects *hardware computing entity*.

Control unit in the CPU controls the running of arithmetic unit according to the value of the data in the memory. For example, in the Intel CPU, if *CPL* bit in the *EFLAG* register not equal to 0, the arithmetic unit may not execute the *SETGDT* instruction. Because data in the memory (*EFLAGS* as in Intel CPU) contains the mask bit of interrupt, it also affects the selection of interrupt events to be handled in the next step.

In order to analyze the essence of the software running in a computer, we give an abstract description for the computer software and hardware as follows.

1. The computer consists of memory, hardware computing entities, and control unit.
2. There are code of a collection of instruction sequence and data objects in the memory.
3. The number of clock cycles for different instruction varies. In the end of a clock cycle, some instruction happen to be completed exactly, others are not and they will continue to execute without disturbing in the next clock cycle.
4. In the end of each clock cycle, there is zero or positive number of event arrived to the computer. According to the current value of each memory location and the hardware computing entities that have just finished an instruction, the control unit selects a number of events and corresponding instruction sequences to be executed in these hardware computing entities from the beginning of the next clock cycle.
5. The execution of each computing entity change only the data in the memory.

As depicted in Figure 2.1.1, the running of an advanced computer may be described as the process of several computing entities operating on a set of data objects in parallel.

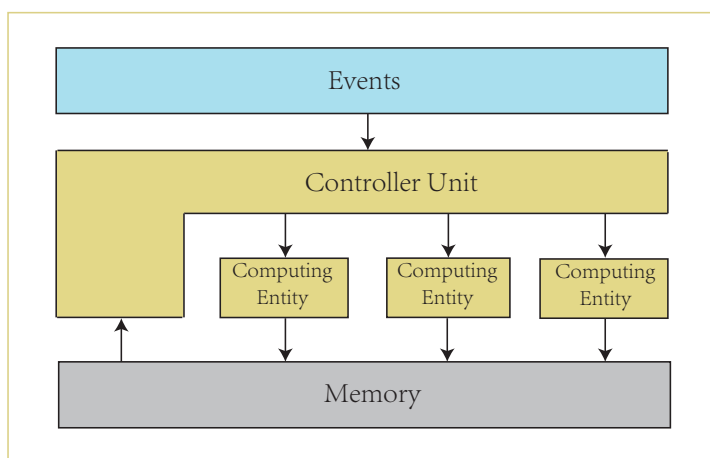


Figure 2.1.1: A group of computing entities operate data in the memory in parallel

Form the software point of view, we can suppose that the hardware computing entities work in an invariable manner. Therefore, the factors affecting the system running are the data and instruction codes in the memory and the arriving events. We have sound reasons to regard a computer as a finite state machine. (1) The set of the accepted events by a computer corresponds to the alphabet of a FSM. (2) The handled events during the time from the power on to the halt instruction correspond to a sentence

accepted by the FSM. (3) At the end of a clock cycle, the value of each memory location and events arrived to the system correspond to the current state of the FSM. (4) A memory location modification action by a number of hardware computing entity corresponds to state transition of the FSM. (5) A state that makes the CPU to execute halt instruction next clock cycle corresponds to termination state of the FSM. A detailed definition of the FSM will be given later in this section.

In the real system, the numbers of objects are finite, e.g., the numbers of child processes created, the files opened. The set of values that an object takes has finite elements. The system state space is the Cartesian product of the sets of values of all objects, and it is a finite set. Theoretically, a process can create any number of objects dynamically and thus the state space cannot be finite. An OS always limits the resources a process may possess. For example, the number of open files for a process is limited, and existing processes in a system is limited.

2.2 OS Architecture

The operating system is the key part of the software system of a computer. The operating system controls any exploitation of the resources of the computer. The operations of an application can be divided to two classes. One class of operation accesses the memory location in its own address space. The other class of operation is sending service request to the operating system such as file services, network services etc and in the period of operation of the service the operating system is in working actually.

Our goal is to design and verify the operating system using a formalized method. We will prove correctness of the operating system implementation. As device drivers and other service program locate outside of the microkernel of the OS, the microkernel OS works in a sealed condition. In other words, each module of a microkernel OS execute as an independent process and their code keep unchanged. Therefore any application cannot modify any data and code of the operating system. And we will prove that the operating system will not interfered by applications in its working time. Before doing this, we construct an operating system model using concept of FSM.

2.3 Independent computing entity

As mentioned above, the value of each memory location determines the selection of instructions to be executed in next clock circle and affects the execution result of these selected instructions. Therefore, the value of each memory location is the key part of the system state. As stated in Sec. 2.1, there are several hardware computing entities in computer that change the state of the system at the same time. We call a sequence of instruction which can run in a hardware computing entity as a **software computing entity**. The process, the thread, the function and a consecutive instruction in the computer system are examples of software computing entity. First of all, we analyze how a software computing entity affects the system state. That is, in which way a software computing entity changes the system state in parallel.

Suppose U is a software computing entity, if the object set $X = \{x_1, \dots, x_t\}$ is its read only data object, we denote it as $R(U)$. If the objects in the set $Y = \{y_1, \dots, y_m\}$ is modified by U during its execution, we denote it as $W(U)$. We call the object set $Z = \{z_1, \dots, z_n\} = \{x_1, \dots, x_t, y_1, \dots, y_m\}$, where $n = t + m$, a **working object set** of U , and we denote it as $RW(U)$. Sometimes the change that a C language function makes to the system state can be expressed as $S' = f(S)$. This expression means that the value of the object set Z , whose value is S when the C function begins to execute, determines the value of the object set Z at its finish time, whose value has been changed to S' . We call a software computing entity having this property as **independent computing entity**. However, in many cases, the contribution that a C language function makes to the change of the system state cannot simply be expressed by a mathematical function. Specifically, the contribution to the system state transition by a system call of Linux generally cannot be expressed as a mapping from the system state that at the beginning time of the system call to the system state at the finish time, even if in a single-core CPU. Because a preemptive kernel may interrupt the execution of system call, the effects that the interruption makes may affect the effect of the successor of the system call. So the contribution that the system call makes to the system state transition is uncertain.

If a software computing entity affects the system state with uncertainty, then it is extremely difficult to verify the correctness and security properties of a system.

There are sound reasons to support that we can separate the system call into several independent computing entities. The fact that seL4 has design its kernel by using functional programming language Haskell may support this idea well. Therefore, we also design and implement the operating system in functional programming style although we implement the OS with C language. We hold the view that if we apply for the resources in a relative concentration way when handling system call or using locking mechanism in a more proper way, we can meet the demands. Whenever we create a new independent computing entity U , we always check whether for any current independent computing entity V the condition $RW(U) \cap RW(V) = \emptyset$ is satisfied. In other words, we design the OS such that it is a composition of a group of independent computing entity during its running time. And we will take advantage of Isabelle/HOL to prove this opinion.

In order to express the system state, which is a complex concepts, in a clear way, we assume that the system service process in our micro-kernel system do not share memory with each other, despite the fact that memory sharing does not prevent us from getting the results we want.

Because of interrupt mechanism, an independent computing entity U may be interrupted after staring execution, it will wait in the waiting queue. During the period of U 's waiting, any other independent computing entity will not access its working object set. We can divide all the current independent computing entity in the system into two types. One is **running independent computing entity** denoted as A_r , which has got its hardware computing entity. The other is **waiting independent computing entity** denoted as A_w , which has begun to execute, but hasn't finished and doesn't possess its hardware computing entity now, thus in the waiting queue. All the independent computing entity set is denoted as A , and $A = A_r \cup A_w$ holds.

2.4 The system object state and system memory state

Here we use the notation in [?] to describe the system state transition. We introduce some notations as $A_r = \{a_1, a_2, \dots, a_t\}$, $A_w = \{a_{t+1}, a_{t+2}, \dots, a_n\}$, and the a_i 's working object set $RW(a_i) = \{x_{ij} \mid j = 1, 2, \dots, n_i\}$, $i = 1, 2, \dots, n$. We denote the value range of object x_{ij} as V_{ij} . The initial values of the object x_{ij} is $[x_{ij}]_s = s_{ij}^0$, $j = 1, 2, \dots, n_i$, $i = 1, 2, \dots, n$. We suppose the running of a_i needs m_i instructions, $i = 1, 2, \dots, n$. After running k instructions, the value of the object x_{ij} in $RW(a_i)$ will be s_{ij}^k , $k = 1, 2, \dots, m_i$, $j = 1, 2, \dots, n_i$, $i = 1, 2, \dots, n$. The value range of the working object set $RW(a_i)$ is

$$\prod_{j=1}^{m_i} V_{ij} \quad (2.4.1)$$

which we denote as $VRW(a_i)$. The independent computing entity a_i 's semantics may be expressed as mapping $\prod_{j=1}^{m_i} V_{ij} \rightarrow \prod_{j=1}^{m_i} V_{ij}$.

We call the Cartesian product

$$\prod_{i=1}^n \prod_{j=1}^{m_i} V_{ij} \quad (2.4.2)$$

the **system object state set**.

Sometimes, an independent computing entity a_i may create new object or release old object in its execution process. In this case, its working object set $RW(a_i) = \{x_{ij} \mid j = 1, 2, \dots, n_i\}$ may vary and so is its value space $\prod_{j=1}^{n_i} V_{ij}$. The value space of the working object set $RW(a_i)$ is actually $\bigcup_{i=1}^n \prod_{j=1}^{n_i} V_{ij}(t_i)$, where $t_i = (k_1, k_2, \dots, k_{n_i})$ and $\prod_{j=1}^{n_i} V_{ij}(t_i) = \prod_{j=1}^{n_i} \underbrace{V_{ij} \times \dots \times V_{ij}}_{k_j}$. In other words, the working object

set $RW(a_i)$ has n_i kind of object, and there are k_j objects of the j -th type in $\prod_{j=1}^{n_i} V_{ij}(t_i)$. We denote the value range of the vector t_i as T_i . Using the notation in type theory [?], the $VRW(a_i)$ can be expressed as

$$\prod (T_i, \prod_{j=1}^{n_i} V_{ij}) \quad (2.4.3)$$

System object state set in the form (2.4.2) may be revised accordingly as

$$\prod_{i=1}^n \prod (T_i, \prod_{j=1}^{n_i} V_{ij}) \quad (2.4.4)$$

For simplicity, we use the term (2.4.1) and (2.4.2) rather than (2.4.3) and (2.4.4) if the context is clear.

In order to express the system object state in Isabelle/HOL, we propose another form of expression of the system object state. First, in the computers memory, in addition to the object in the (2.4.2), there are a lot of free storage space fragment. The positions and number of these fragments are constantly changing. Second, if using (2.4.2) to describe the system object state, we need to introduce a lot of function object in Isabelle/HOL, which would introduce unnecessary complexity.

We make unified numbering for all the memory units including register, cache, memory, etc. These numbers as a whole is a subset of the natural numbers $\mathbb{N}$, and we denote this subset as $\mathbf{N} = \{0, 1, 2, \dots, N\} \subset \mathbb{N}$.

Each data object has its type, occupies one or a number of several consecutive memory locations. For example, a 32 bits integer variable y occupies storage units $\{\&y, \&y + 1, \&y + 2, \&y + 3\}$ which is four consecutive storage unit. We will use interval $[\&y, \&y + 3]$ to express the set of this storage unit. Here we call it **subinterval** of $\mathbf{N}$. If the current system has n processes $p_i, i = 1, 2, \dots, n$, and process p_i 's working object set is $\{x_{ij} | j = 1, 2, \dots, n_i\}$, and these objects occupy the mutual disjoint subintervals $N_{ij}, j = 1, 2, \dots, n_i, i = 1, 2, \dots, n$ of $\mathbf{N}$. Besides the subintervals occupied by the objects $\{x_{ij} | j = 1, 2, \dots, n_i, i = 1, 2, \dots, n\}$ in $\mathbf{N}$, we suppose there are still n_0 disjoint subintervals left not occupied and we denote them by $N_{0j}, j = 1, 2, \dots, n_0$. And the corresponding data objects are denoted by $x_{0j}, j = 1, 2, \dots, n_0$. It is obvious that the subintervals $N_{ij}, j = 1, 2, \dots, n_i, i = 0, 1, 2, \dots, n$ are disjoint, and $\cup_{i=0}^n \cup_{j=1}^{n_i} N_{ij} = \mathbf{N} = \{0, 1, 2, \dots, N\}$. Without lose generality, we may assume that the value at the unoccupied memory location $N_{0j}, j = 1, 2, \dots, n_0$ are 0. The reason is that all the value of all newly allocated data object must be 0 in order to ensure the system has the property of no object reusing.

We denote the object, which occupies only i -th storage, as $y_i, i = 0, 1, 2, \dots, N$. The values of these kind of objects are called a **system memory state**. We denote the system memory state by $s, s_1, s_2, \dots$, etc. The value of the object y_i in the state s is denoted by $[y_i]_s$. Clearly, the system memory state s can be viewed as a mapping of $\mathbf{N}$ to $\mathbb{N}$. i.e.

$$s : \mathbf{N} \rightarrow \mathbb{N}$$

that is $[y_i]_s = s(i)$.

If the storage space occupied by the objects $\{x_{ij} | j = 1, 2, \dots, n_i, i = 1, 2, \dots, n\}$ is subintervals $N_{ij}, j = 1, 2, \dots, n_i, i = 1, 2, \dots, n$ of $\mathbf{N}$, then the value of these objects, which is a system object state, determines the mapping $s : \mathbf{N} \rightarrow \mathbb{N}$, because the remaining subintervals are $N_{0j}, j = 1, 2, \dots, n_0$ and they are unoccupied and they all have the value 0. On the other hand, if location of all working object set of all processes $p_i, i = 1, \dots, n$, has determined, the mapping $s : \mathbf{N} \rightarrow \mathbb{N}$ determines the system object state.

2.5 Events

During each clock cycle, one device may produce interrupt and it is possible to change set A_r . The exception of CPU may also change set A_r . A process may execute a software interrupt instruction and changes set A_r in result. We call the device interrupt, exception and software interrupt produced during a clock cycle **arriving event** which is denoted as E_a . The events which has arrived before and is waiting to be handled are called **waiting events** denoted as E_w . The system object state, current arriving events E_a , running independent computing entities A_r , waiting independent computing entities A_w , and

waiting events E_w determine next running events E_r and waiting events E_w . It also determines running independent computing entities A_r and waiting independent computing entities A_w .

2.6 OS finite state Model

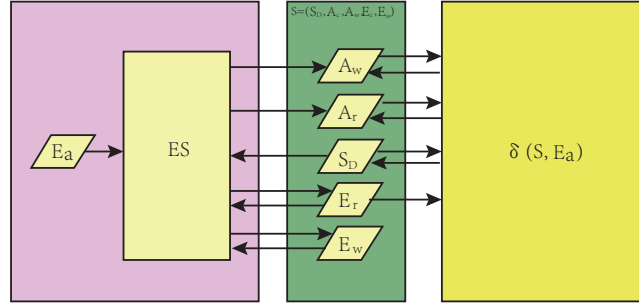


Figure 2.6.1: FSM model

Base on the above analysis, we build the operating system finite state machine model as follows.

Definition OS FSM model is a FSM $A_{OS} = (S, \Sigma, \delta, s_0, \Gamma)$, and the meaning of $S, \Sigma, \delta, s_0, \Gamma$ is as follows:

- (1) State $S = (S_D, A_r, A_w, E_r, E_w)$
 - a) The object state S_D : the system object state.
 - b) The running independent computing entities A_r : the independent computing entities are now running in a hardware computing entities;
 - c) The waiting independent computing entities A_w : the independent computing entities which are waiting for a hardware to run;
 - d) The running events E_r : the events are chosen to be handled;
 - e) The waiting events E_w : the events are recorded to wait to be handled;
- (2) Σ is the set of all kind of events the system accepted.
- (3) The transfer functions $\delta : \delta(s, E_a) = s'$.
- (4) s_0 is the original state of system.
- (5) $\Gamma \subseteq S$ is the termination state set. Whenever the system reaches a state in Γ , then the system terminate.

3 Verification method of an OS in Isabelle/HOL

3.1 Principle of Isabelle/HOL Theorem Proving

Isabelle/HOL has predefined a series basic types. User can also define new types by keywords *record* and *datatype*. One can define data objects by keyword *definition*, define function objects by keyword *fun* and *primrec*, and define propositions by *lemma* and *theorems*.

A domain is established by composing a theory in a text with Isabelle/HOL. In general, each theory is a collection of types, definition, functions, theorems and proofs. The following listings describe a general form of a theory and a proof.

Listing 3.1.1: Theory structure

```

Theory T
  Imports T1, T2, ..., Tm

  Begin
    types, definition,
    functions, theorems
    proofs
  end

```

Listing 3.1.2: Proof structure

```

proof
  assume formula 0
  have formula 1 by simp
  .....
  have formula n by blast
  show formula n+1 by .....
  .....
qed

```

Isabelle/HOL provide many rules to prove theorems. Meanwhile, we can quote theories which contain one or more proven theorems by the keyword imports. Having import a theory, we can prove new theorems by using the proven theorems in the imported theories.

The interactive reasoning process is like this: first we define a model using types, definition, functions and theorems, then we can use apply to tell Isabelle/HOL to prove a theorem. Sometimes you are lucky when Isabelle/HOL tells you that the theorem is proved. It is of common occurrence that Isabelle/HOL only gives you several theorems, which are sufficient to prove the target theorems, to be proved. Therefore, Isabelle/HOL provides a structure proof language Isar. A typical Isar proof structure is the proof-qed structure. In a proof-qed structure, it point out that in order to prove a target theorem what formula we must assume and what reasoning rules we suggest using. By using Isabelle/Isar/HOL, we can prove a theorem more automatically.

3.2 Domain of OS

As described above, it is obvious that the system object state can be represented as an element of a domain $[?, ?]$, which is a mathematical system. A domain $\mathbf{M}$ consists of three parts.

The first part is non-empty set of elements M , and it can be used to describe the set of all possible values that an object, which is an element of the working object set of a process, may take. It is clearly that the set $\prod_{i=1}^n \prod_{j=1}^{n_i} V_{ij}$ is subset of M .

The second part is a non-empty set of functions on M . The semantics of a function in implemented software can be expressed by a mapping from a group of values that is the working object set take at beginning time of the function to another group of values which the working object set take at finish time of the function. It turns out that the functions on M can be used to express the semantics of the function in software. The functions on M can express how a computing entity using values the working object take at the beginning calculates new values the working object take at the end.

The third part is a non-empty set of propositions about M . The propositions about M can be used to present the current properties of a sequence of instructions, a process or the whole system. Some propositions about M only contain the elements of M . They describe the properties of the system at the beginning time of an operation. Some propositions about M also contain the function on M , so they describe the properties of the system at the end of an operation because the function represents the system state transition then. Some propositions about M describe the properties pertaining to the system all the time. For example, that fact that all processes have their independent memory space can be expressed as a propositions about M that working object sets of any two processes are non-intersect at any time.

Isabelle/HOL is an interactive theorem prover. It has implemented a logical reasoning tool. It has also defined a syntax by which one can describe the objects, the functions operating on objects, the proposition about current objects and functions etc. So it can creates a domain $[?]$ or a structure $[?]$ denoted as $\mathbf{M}_{Isabelle/HOL}$. As described in section 2.1, the current system state, operation of the hardware computing entity, system properties we are concerned with constitute a domain denoted as $\mathbf{M}_{Computer}$.

In order that a proposition proved to be true in $M_{Isabelle/HOL}$ if and only if the corresponding proposition in $M_{computer}$ is true, we should define a concept of isomorphism.

Definition Let $\mathbf{M}_1 = (M_1, F_{M_1}, P_{M_1})$ and $\mathbf{M}_2 = (M_2, F_{M_2}, P_{M_2})$ be two domains. If there exists a mapping $f : M_1 \rightarrow M_2$ satisfying the following:

- (1) $f : M_1 \rightarrow M_2$ is an one-to-one mapping;
- (2) there exists an unique mapping $g : F_{M_1} \rightarrow F_{M_2}$ such that for any $c \in F_{M_1}$, $\delta = g(c)$ is defined as $\delta(f(x)) = f(c(x))$ for any $x \in M_1$;
- (3) there exists a unique mapping $h : P_{M_1} \rightarrow P_{M_2}$ such that if h maps a predicate $p(x_1, x_2, \dots, x_n)$ in P_{M_1} to $q(y_1, y_2, \dots, y_n)$ in P_{M_2} , then $p(x_1, x_2, \dots, x_n)$ is true if and only if $q(f(x_1), f(x_2), \dots, f(x_n))$ is true.

we say that the domain M_1 is isomorphism to M_2 .

Because the logical formulae are constructed by constants, variables, functions, terms and logical conjunctions, so if M_1 is isomorphic to M_2 , and it is easy to see that a proposition p in P_{M_1} is true if and only if its corresponding $q = h(p)$ is true. From the definition above, if M_1 is isomorphic to M_2 , then the mapping $f : M_1 \rightarrow M_2$ determines the mapping g and h uniquely. Therefore in order to construct the domain $\mathbf{M}_{Isabelle/HOL}$ such that it is isomorphic to $\mathbf{M}_{computer}$, we only need to construct a one-to-one mapping $f : \mathbf{M}_{Isabelle/HOL}.M \rightarrow \mathbf{M}_{computer}.M$ and based on the mapping f to derive the mapping $g : \mathbf{M}_{Isabelle/HOL}.F_M \rightarrow \mathbf{M}_{computer}.F_M$ and the mapping $h : \mathbf{M}_{Isabelle/HOL}.P_M \rightarrow \mathbf{M}_{computer}.P_M$.

We construct the domain $\mathbf{M}_{Isabelle/HOL}$ according to the design and implementation of OS, so there is isomorphism [?] from $\mathbf{M}_{Computer}$ to $\mathbf{M}_{Isabelle/HOL}$. Because the logic system of is reliable and complete, so we can use the logical reasoning tool of Isabelle/HOL to prove the properties of the OS.

As shown in the figure 3.2.1, we establish the relationship between the OS and Isabelle/HOL logical reasoning system. First we construct a OS FSM model and design and implement an OS according to the model and construct a domain $\mathbf{M}_{computer}$ accordingly. Then we construct a domain in Isabelle/HOL which is isomorphic to the $\mathbf{M}_{computer}$. Thereafter, the properties about the function or security of the OS can be mapped to logic formulas. On this basis, we can prove the properties of an OS through the reasoning or the proving in the logic system.

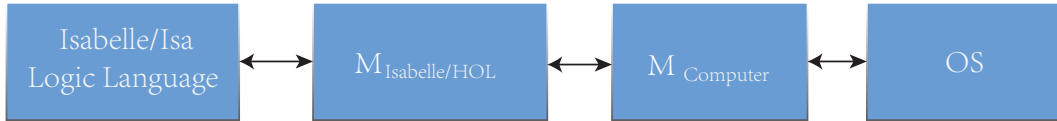


Figure 3.2.1: Relationship among the OS, domains and Language

3.3 Designing an OS as finite state machine

What is an operating system? We regard OS is a server that provides its service whenever a user program claims a request. After finishing its service, the OS gets ready to receive another request. As described in section 2.6, an OS works like a FSM $A_{OS} = (S, \Sigma, \delta, s_0, \Gamma)$. Now we explain how the OS maps to a FSM.

The principal elements of the OS FSM $A_{OS} = (S, \Sigma, \delta, s_0, \Gamma)$ are state set S and state transition function δ . An element of S is a vector which is the composition of the values of all the data objects of all current processes. All the event handle functions constitute state transition function δ . What an event handle function does is that it transits the data objects from initial values to values required by semantics.

Designing a OS domain, there are also two principal works, designing of the working object set and designing of the event handle functions which operate on the working object set. The processing of events in operating system corresponds to the transition of value of working object set.

Therefore, in order to prove the OS accomplishes all its function correctly, we should prove that the OS correctly transits the data objects from initial values to values required by semantics whenever it handles an event. In order to prove the security of an OS, we will prove the system state satisfy the security condition in any time.

Now we explain how we design the microkernel OS VTOS as an FSM. Firstly, we design the working object set $\{x_{ij}|j = 1, 2, \dots, n_i\}$ for all module M_i . Suppose the object x_{ij} takes value on set $V_{ij}, j = 1, 2, \dots, n_i$. Then the possible states for the module M_i is the space $\prod_{j=1}^{n_i} V_{ij}$. Then we design all handle function set $\{f_{ij}|j = 1, 2, \dots, m_i\}$ for module M_i . The function $\{f_{ij}|j = 1, 2, \dots, m_i\}$ set is part of the components of the transition function δ , which is obviously a piecewise defined function. The modules of the VTOS are microkernel, file manage management system, process management system, memory management system, device drivers.

The verification contains correctness and security properties. We will take microkernel for example to explain the method of designing and verification in this paper.

3.4 Proving the correctness of a function in Isabelle/HOL

As described above, the semantics of a function can be expressed as a mapping from state to state. In order to prove the correctness of a implemented function f in C language, we can prove that for any starting state for f it transits the state to the required state according to the semantics. There are two task should be accomplished to prove the correctness of a function.

The first task is to express state transition function f . A function in C language has two aspects, which are the working object and the instruction sequence. Firstly, we define the working object in Isabelle/HOL, and there exists a isomorphic mapping between the working objects in C language and data objects in Isabelle/HOL. In order to translate the instruction sequence of C code to code in Isabelle/HOL, we translate the C code to an assembly code, and then we translate each assembly code to a function in Isabelle/HOL.

The transforming C code to assembly code is done by the compiler. That is, we let compiler output the assembly codes. It can be hold that the assembly codes are consistent with the machine codes of the program running in the hardware platform. So we do the verification on the assembly level and thus proving the consistency between the C code and binary code of the program is unnecessary. Previouslywe translate each instruction in the assembly codes to the corresponding objects in the model of Isabelle/HOL one-by-one.

As shown in Listing 3.4.1, we define a type as Instr, which contains all kind of assembly code, such as mov, add, push, pop, leave etc. The object “movrr Register Register”, whose type is Instr, corresponds to the assembly code “movl Register Register”. The function PCinc adds 1 to PC register. The semantics of the function NextS is that the instruction Instr, which is the second parameter, transits the starting state, which is the first parameter, to another state, which is the third parameter. At last, a assembly code is translated such that the function NextS applies to the current state s and a Instr object, which corresponds to the assembly code. The assembly code “movl register, register” is defined as the last line of the Listing 3.4.1. There are two reasons that we translate the C code to an assembly code. The assembly code is exactly equal to the machine code, therefore we need not to solve the correctness problem of a compiler. We can also verify that whether an interrupt, which is occurred between two assemble code, will affect the execution result of the function. By applying the sequence of Isabelle/HOL functions, the stating state is transited to a new state in Isabelle/HOL. A instruction in an OS or in Isabelle/HOL is an object, it occupies one memory position in Isabelle/HOL but generally several consecutive position in an OS. All instructions in a module constitute a object set. We call the instruction set of microkerne of VTOS as *KERNEL_INST*, the instruction set of process manager as *PM_INST*, etc. The second task is to figure out a logic formula to express the relation between the two states which is accordant with the semantics of function f .

Listing 3.4.1: Isabelle/HOL function for a assembly code

```
datatype Instr=
  | movrr  Register Register
  | movir  int      Register
  | addrr  Register Register
  | .....
```

```

fun PCinc::" state=>state"
where
  PCinc s = s ⋈ R:=((R s) (pc:=((R s) pc)+1))⋈

primrec NextS::" state=>Instr=>state"
where
  NextS s (movrr y x) = PCinc (s ⋈ R:=((R s) (x:=((R s) y))) ⋈ )
  .....

```

3.5 Proving the Integrity property in Isabelle/HOL

A running OS is a collection of several processes. Each process provides designated services. Whether these processes can preserve the same services while the computer is working? Whether the processes of the OS can defend itself from attacking? How to describe and judge that a process of the OS preserves its services? These question confused many researches.

We regard that an OS corresponds to a finite state machine, and we prove the immutability of the corresponding finite state machine of the OS to prove the integrity of the OS.

We mentioned that the current values of all working objects of the processes of the OS correspond to the state of the finite state. The functions and their relevant working objects of the OS constitute the transition function. It is obvious that if the accepted alphabet and transition function of a finite state machine remain unchanged the machine is immutable. Therefore we need only to prove the accepted events and the semantics of all functions of the OS, which are correspond to the accepted alphabet and transition function of the finite state machine, are kept unchanged.

In order that we may prove the invariance of the semantics of all functions of the OS while running more clearly, it is designed that all working objects of any two processes do not intersect, and that the possible target address set of any branch point in any function is fixed and cannot be changed by user input. Because the services provided by an OS is fixed, this objective is reasonable. That we has followed these criteria to design our VTOS shows that it is feasible also. On this premise, we should only prove that all memory locations the OS code occupies remain unchanged, and that the selection of target address in each branch point is consistent with the semantics of the function, because this actually proves that all factors that may affect the integrity of the OS are kept unchanged.

One may worry about that it is possible to attack the application system successfully by using a series function of the OS without modifying the function of the OS. It is the responsibility of access control policy to defend the attack using combinational function of the OS. For example, an attacker may use the vulnerability of the program IE to elevate the privilege of another program. But if we control the capability of the program IE successfully, we may prevent from elevating the privilege by using vulnerability of a program. We will discuss this problem later. Here we concentrated on proving the integrity of the functions of an OS.

4 Verification of the microkernel of VTOS in Isabelle/HOL

4.1 Constitution of VTOS

After studying the OS such as Linux, Minix, etc, we design and implement an microkernel OS, which we called VTOS (Verified Trusted Operating System). We have verified the functionality correctness of the microkernel, and still promote the formal work of the other components, such as file system, network management, etc. The microkernel of the VTOS provides the most basic services, such as schedule, interrupt handling, message service and simple I/O service. Other services including file management system, process management system, etc, are implemented as user mode processes. The architecture VTOS is shown as the following figure 4.1.1.

The principal reason for us to choose the microkernel architecture is that the microkernel is the only code running in highest privileged level and it is also closed. The device drivers are all user mode programs. Any new OS service, if needed, is added in user mode. Any code must not be added into the microkernel,

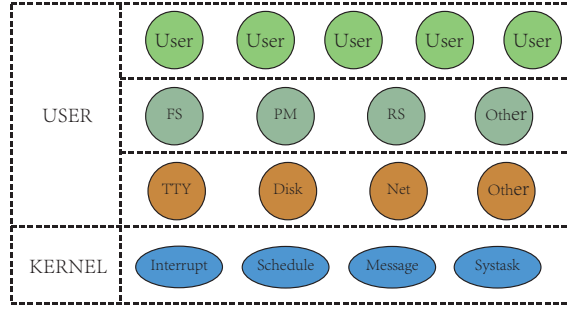


Figure 4.1.1: The architecture of VTOS

therefore a user mode program cannot directly damage the microkernel. It avoids any code including drivers attacking the microkernel and other parts of the VTOS. The microkernel OS is adequate for multi-core or many core platform [?].

In this section, we will introduce our formal design methods for VTOS and formal method of verifying the correctness and security properties of VTOS using the Isabelle/HOL. We will describe the formal design and verification method for other service processes, such as process management, memory management, file management, device management in the VTOS in other papers.

4.2 Modeling the microkernel of VTOS in Isabelle/HOL for verification

While creating a model or a structure of a logic language, the key problem is to construct the domain and define the mapping or interpretation from logic language to the domain. Also in order to verify the correctness and security properties of an OS, the key problem is also to create an OS domain in Isabelle/HOL. While we design the working object set for a module of the VTOS, we have designed the state set of a FSM actually. And while we design the system call function, we have designed the state transition function for the FSM actually. While we establish the correctness and security condition, we have constructed the relations on domain actually. So when we complete the OS design, we also complete the construction of the domain. We translate these components to a domain in Isabelle/HOL through the assembly code of the VTOS. And the two domains are isomorphic, so it becomes the basis of verification of the VTOS. In this section we will describe the process in detail.

Defining the data object in Isabelle/HOL

The microkernel of VTOS has defined its working object set including gdt which is the global description table, idt which is the interrupt description table, proc_table which is the array of process control table, ready_head which is the head of schedule queue of processes, ready_tail which is the tail of schedule queue of processes, etc.

In the VTOS, we create interrupt descriptor table idt as depicted in Listing 4.2.1. Accordingly, we define the corresponding object idt_bass in Isabelle/HOL as depicted in list 4.2.2. We define the system state in Isabelle/HOL by function $\mathbf{N} \rightarrow \mathbf{N}$. An object occupies one or several location of the memory, and we identify an object by the interval of its memory locations. So, we define a object in Isabelle/HOL with a constant representing its address in memory. The constant idt_bass in 4.2.2 represents the object idt . The i -th interrupt descriptor can be identified by the memory interval $[idt_bass + (i - 1) * size_idt, idt_bass + i * size_idt - 1]$.

Listing 4.2.1: IDT descriptor

<pre> struct GateDescriptor { uint_32 offset_15_0 : 16; uint_32 segment : 16; uint_32 pad0 : 8; uint_32 type : 4; uint_32 system : 1; </pre>	<pre> uint_32 privilege_level : 2; uint_32 present : 1; uint_32 offset_31_16 : 16; }; struct GateDescriptor idt []; </pre>
--	--

Listing 4.2.2: idt bass

```
idt_bass
definition idt_offset_low::"int"
where "offset_low==0"

definition idt_segment::"int"
where "idt_segment Idt == 1"
```

```
define idt_flag::"int"
where idt_flag == 4

definition offset_high::"int"
where "offset_high == 1"
```

In C language, we access a component of a interrupt descriptor by its name, for example $idt[i].segment$ indicates the segment selector of the interrupt handler. In Isabelle/HOL, we define corresponding constant for each component with its offset in the interrupt descriptor, such as `idt_offset_low` for component `offset_15_0`, `idt_segment` for component segment, etc as described in Listing 4.2.2. In Isabelle/HOL, an object is identified by its memory location. Therefore, when accessing a component of an interrupt descriptor, we use the beginning location of the interrupt descriptor and the constant corresponding to the component. For example, the component $idt[i].segment$ in C language is denoted as $idt_bass + (i - 1) * size_idt + idt_segment$ in Isabelle/HOL.

Defining a function in Isabelle/HOL

Here we take initialization of interrupt descriptor table for example to explain the method of defining a function in Isabelle/HOL. We use `init_segment()` and `set_segment()` to initialize the GDT, and use `init_idt()`, `set_trap()` and `set_intr()` to initialize IDT. After initialization, IDTR register points to the object `idt[]` which takes the designated value as table 4.2.1 describes.

Table 4.2.1: idt

63	48	47	46	45	44	43	40	39	32	31	16	15	0
offset 31:16	p	DPL	s	type	pad	segment select	offset 15:0						
0xc010	0	0x0	0	0xF	0x0	0x8	0x0d44						
...	...	...	...	...	...	...	...						
0xc010	0x1	0x0	0x0	0xE	0x0	0x8	0x0dea						
...	...	...	...	...	...	...	...						
0xc010	0x1	0x3	0x0	0xF	0x0	0x8	0x0d2c						
...	...	...	...	...	...	...	...						

After the OS has been loaded into the memory, all the objects has value as program designated. At this time, as the object `idt` is just defined without assigning values, it is the responsibility of the initialization module to assign initial values.

The initialization code for the *idt* in C language and assemble are shown in list 4.2.3 and 4.2.4. In order to translate assemble code to instruction in Isabelle/HOL, we create a library, in which each assemble instruction is designed as a function in Isabelle/HOL. For example, the function `movrr` in Isabelle/HOL is implemented for the assemble instruction with the form of *movl reg, reg*. The initialization function for *idt* is translated to a sequence of function in Isabelle/HOL which is described in list 4.2.5.

Listing 4.2.3: initialization code in C

```
void set_trap(
    struct GateDescriptor *ptr,
    uint_32 selector,
    uint_32 offset, uint_32 dpl) {
    ptr->offset_15_0 = offset;
    ptr->segment = selector;
    ptr->pad0 = 0;
    ptr->type = TRAP_GATE_32;
    ptr->system = FALSE;
    ptr->privilege_level = dpl;
```

```
ptr->present = TRUE;
ptr->offset_31_16 = (offset
    >>16)&0xFFFF;
```

Listing 4.2.4: initialization code in ASM

```
push    %ebp
mov     %esp, %ebp
```

...	mov 0x10(%ebp), %eax	mov 0x8(%ebp), %eax
mov %eax, %edx		mov %eax, %edx
		...

Listing 4.2.5: initializaiton code in Isabelle/HOL

```

definition set_trap::"Code"
where
  "set_trap =
    pushr    ebp;
    movrr    esp    ebp;
    ...
    movirr   4      ebp    eax;
    movrr    eax    edx;
    movirr   2      ebp    eax;
    movrir   edx
    ..."

```

In Isabelle/HOL, the function *movirr n reg1 reg2* means that moves the value in memory location *reg1 + n* to *reg2*. During the execution of *set_trap* in Isabelle/HOL, the *ebp + 4* point to the parameter offset. Therefore, the function *movirr 4 ebp eax* moves the parameter offset to register *eax*. After executing the *set_trap*, the object *idt* in Isabelle/HOL *idt_bass* is assigned the value as required.

After defining the data objects and defining functions in Isabelle/HOL, we establish a logic model in Isabelle/HOL which is isomorphic to the model of FSM for VTOS. A logic formula in Isabelle/HOL is mapped to a property in the VTOS. Therefore we can prove a property in VTOS through proving a assertion in Isabelle/HOL.

4.3 Proving the correctness of OS initialization in Isabelle/HOL

Here we take the correctness proof of the initialization function of the `idt` for example to explain method of proving the correctness of a function in Isabelle/HOL. We defined an object *state*, which is described in list 4.3.1. `M` is a component of the object of *state*, which is a function of the type $nat \rightarrow nat$ mapping each memory location number to the value it holds. Other components of the object of *state* are *reg*, which represent the registers etc.

Listing 4.3.1: define object

```

record state=
  ...
  runseg::"nowseg"
  Mem::"int set"
  M::"int  $\Rightarrow$  Memcell"
  gdt::"int"
  idt::"int"
  ...

```

Listing 4.3.2: verify the object

```

...
s.M(idt_base+offset_low) = addr_low
s.M(idt_base+idt_segment) = segment

s.M(idt_base+idt_flag) = flag
s.M(idt_base+offset_high) = addr_high
...

```

After the execution of the initialization function as described in previous subsection, we should verify whether the working objects have been assigned the right values. In order to do this, we need to establish a series of logic formula expressing the conditions that the initialization function has correctly assigned the value to the working object set. In the list 4.3.2, s is the state object, and $s.M$ represent the memory function component, and $s.M(idt_base + offset_low)$ represent the value in the memory location $idt_base + offset_low$. The formula $s.M(idt_base + offset_low) = addr_low$ means that the value in the memory location $idt_base + offset_low$ has been assigned as $addr_low$ at the checking time.

4.4 Proving the Event Handling and State Transform of VTOS Microkernel

Whenever an event occurs, the OS will select an event handler to execute. The event may be an interrupt, sending message request etc. It is important to prove the correctness of the execution of event handler. Here we take the example of message module to describe the proving method.

In the microkernel of the VTOS, the message service module contains the function *send*, *receive*, etc. The working object of these function are some components of the two objects of PCB (process control block), which is described as table 4.4.1, of the message sending and receiving processes.

Table 4.4.1: object using by message mechanism

field	type	comment
...	...	...
p_rts_flags	int	Receiving, Sending, None
p_getfrom	int	Processid from whom this process want to receive
p_messbuf	message*	pointer to the buffer used to deposit waiting message
p_sendto	int	Processid to whom this process want to send
p_caller_q	proc*	head of list of members of procs wishing to send msg to this process.
p_q_link	proc*	link to next proc wishing to send
...	...	...

Here we explain the prove process of the function of *sys_send* in VTOS, which send message for a process to another process. The main part of the C code of the function *sys_send* is described in Listing 4.4.1, assemble code in Listing 4.4.2, Isabelle/HOL code in Listing 4.4.3.

Listing 4.4.1: *sys_send* in C

```
int sys_send(struct proc *caller_ptr, int dst, message *m_ptr, unsigned flags)
{
    struct proc *dst_ptr = get_proc_from_pid(dst);
    .....
    copy_mess(m_ptr, dst_ptr->p_messbuf, sizeof(message));
    .....
}
```

Listing 4.4.2: *sys_send* in ASM

```
<sys_send>:
push    %ebp
movl    %esp,%ebp
subl    $0x28,%esp
movl    0xc(%ebp),%eax
movl    %eax,(%esp)
call    c0101128
.....

movl    %eax,0x4(%esp)
movl    0x10(%ebp),%eax
movl    %eax,(%esp)
call    c010116d <copy_mess>
.....
```

Listing 4.4.3: *sys_send* in Isabelle

```
definition sys_send::''Code''
where
  sys_send =
    pushr      ebp;
    movrr     esp ebp;
    subir     10 esp;
    movirr    3 ebp eax;
    movrir    eax 0 esp;
    call      get_proc_from_pid;
    .....
    movrir    eax 1 esp;
    movirr    4 ebp eax;
    movrir    eax 0 esp;
    call      copy_mess;
    .....
```

It is not a trivial task for us to establish the formula to express the condition for the correctness of a state transition function whenever the function is called, because this condition should fit for any starting state for the state transition function. For example, the function *sys_send* sending message for a process

to another process converts the state s to state s' . We set the formula to be proved as

$$\forall s, s' = sys_send(s) \rightarrow P(s, s') \quad (4.4.1)$$

which means starting from any state s , the function sys_send can correctly send designated message to target process. But to prove this assertion, we may need to check that for all the state s this formula is satisfied. We think it may not work in Isabelle/HOL in general case, because there is a large number of state and it is almost impossible to check that whether a formula is true for all the state one by one. We regard that not all the state is security state. Therefore, there are many states to which the OS may not reaches and we need not consider these cases that a function in the OS starts from these states. There is a particular starting state condition what is always satisfied for a function in the OS. We suppose the semantic formula of the function sys_send is $P(s, s')$ and the starting state condition of it is $Q(s)$. We can reach the conclusion that the function sys_send may be regarded as correctly implemented if and only if the following formula is true.

$$\forall s, Q(s) \wedge s' = sys_send(s) \rightarrow P(s, s') \quad (4.4.2)$$

The formula 4.4.1 and 4.4.2 differ substantially, because it is easy to prove the assertion $s' = sys_send(s) \rightarrow P(s, s')$ on the premise $Q(s)$ and it is difficult without the premise. Now we will figure out the starting state condition $Q(s)$ and the semantic formula for sys_send . A function's state condition concerns mainly to its working object set. The message function sys_send 's working object set includes actual parameters in the stack, accessibility of the corresponding components of the process control block object in the object *proc* of sending and receiving process. The formula

$$Q(s) : (s.regsp + 1 = caller_ptr) \wedge (s.regsp + 2 = dst) \wedge (s.regsp + 3 = m_ptr)$$

states that the actual parameters which the function sys_send needs are located in the stack at proper location and possess the proper values. For sake of simplicity of description we omit the condition for accessibility of PCB objects in *proc*. When a process requires to send a message to another process, the target process may just wait for this message or it does not wait for this message. Here we explain the case that the target process is waiting for this message. In this case, the function of sys_send is copy a designated length of bytes from actual parameter to another buffer. Therefore, the correctness of function of sys_send is that the values in the relevant two memory location interval are equal. Before we give the formula $P(s, s')$, we define an auxiliary formula

$$CmpM(s, p, q, n) = (i \geq 0 \wedge i \leq (n - 1)) \rightarrow (s.M(p + i) = s.M(q + i)).$$

At last we reach the semantic formula as

$$P(s, s') = CmpM(s', s.M(s.regsp+3), s'.M(proc+sizeof_proc*s.M(s.regsp+2)+p_messbuf), sizeof_msg)$$

4.5 Proving the integrity of the VTOS in Isabelle/HOL

Here we take the method of verifying the integrity of interrupt handler for example to explain the method of proving the integrity of the microkernel of VTOS. When interrupt event arrives to the system, the controller unit of the CPU decides which instruction will be executed next based on the current state, for the current state tells whether the arrived interrupt is blocked. If the interrupt has not been blocked, the controller unit of the CPU set the program counter register (PC) to the entry of the corresponding interrupt handler according to the current state which includes *IDTR*, *IDT*, *GDTR*, *GDT*, *KERNEL_INST* etc. Therefore, in order to prove the integrity of the interrupt handling, we need to prove that the microkernel always jump to the same correct entry point of the interrupt handler, and to prove the instructions of the interrupt handler would not be modified by any instruction, and to prove the interrupt handler are correctly implemented. So we have three tasks. The third are similar to the method mentioned in section 4.3 and 4.4. Therefore, we only explain the first and second

tasks in this section. In fact, in the microkernel of VTOS the entries and instructions of the existing interrupt handler are not modified at any time. And if this property is proved, it is reasonable to believe that it accomplishes the first and second tasks. So we figure out a logic formula to express this property.

To begin with, we define two functions. First, we define a function *idt_offset* that given an interrupt descriptor it returns the interrupt handler entry corresponding to the interrupt descriptor. Therefore, when an interrupt with index n arrives, the control unit of the CPU set *PC* register *s.regs.pc* as $s.M(idt_offset(s.regs.IDTR + (n - 1) * sizeof_IDT))$ according to current state s . The address of the designated interrupt handler that the control unit sets is decided by the object *s.regs.IDTR*, the *interrupt descriptor* in IDT located in memory location at $s.regs.IDTR + (n - 1) * sizeof_IDT$, and the corresponding segment descriptor and segment. Second, we define the function *Step(s, inst)* to indicate the result state s' to which the instruction *inst* transits from the state s .

We reach our required logic formula as follows.

$$\begin{aligned} \forall inst \in KERNEL_INST \quad s' = step(s, inst) \rightarrow \\ (s.regs.IDTR = s'.regs.IDTR) \wedge (s.regs.GDTR = s'.regs.GDTR) \wedge (s.regs.LDTR = s'.regs.LDTR) \wedge \\ (\forall i \in [0, MAXIDT * sizeof_IDT) \quad s.M(s.regs.IDTR + i) = s'.M(s'.regs.IDTR + i)) \end{aligned} \quad (4.5.1)$$

The formula 4.5.1 means that any instruction does not modify the register *IDTR* and the *IDT*. Similarly, we can figure a formula to express the property that any instruction does not modify the memory area where interruption handler codes locate.

5 Conclusion

In this article, we defined finite state machine model of an operating system. With the help of the model, we can design the OS formally. We construct a domain of an OS and construct a domain in Isabelle/HOL. Because the two domains are isomorphic, to prove a property in OS we only need to prove the corresponding property in Isabelle/HOL. With the help of the OS FSM model, we can also precisely express the concept of correctness of a function. We regard the operating system as a finite state machine. The operating system handles the events arrived to system continuously just like a finite state machine reads the character from a sentence one by one and transits its state accordingly. Therefore, we can also express the properties of operating system in light of the concepts concerning the finite state machine. For example, the condition of the integrity of the interrupt handler is raised upon the concept of finite state machine. Experiments show that translating the assemble code of the operating system into Isabelle/HOL is feasible and practical.

There are still much work to be done to design and verify a trusted operating system. We have to elaborate the designing and verification of the user level service programs of the operating system. We also have to raise and prove logic formula expressing the other necessary security condition of the operating system so that the operating system is trusted. We will continue to do this work in separate papers in short later.

Acknowledgements

This work was supported by the National High Technology Project 863 under Grant Nos.2011AA1A202 and National Science Foundation of China under Grant Nos. 61021062