Send Orders for Reprints to reprints@benthamscience.ae

The Open Cybernetics & Systemics Journal, 2015, *9,* 1125-1129

Open Access

A Formal Specification in B of an Operating System

Chen Danmin[1], Sun Yue[2] and Chen Zhiguo[3,*]

[1]*School of Software, Henan University, Kaifeng, Henan, 475004, P.R China;* [2]*School of Information Engineering, Kaifeng University, Kaifeng, Henan, 475004, P.R China;* [3]*Institute of Data and Knowledge Engineering, Henan University, Kaifeng, Henan, 475004, P.R China*

Abstract: Operating system (OS), as a significant system software, provides services and security protection to a variety of applications. Thus, the correctness of OS is a core issue of computer systems. To ensure the correctness of operating systems, it is a recognized way to use the rigorous formal methods. In this paper, we apply B method to design a formal model of OS called fmC/OS. The process is split into two phases. At first, software document specifications are formalized into an abstract model and then the abstract model is implemented into the concrete model. The concrete model is the base of translating executable code. Operating System based on B method can strengthen validity and security of the system.

Keywords: B method, formal design, operating system, specification.

1. INTRODUCTION

Operating system (OS) is a significant system software as well as the support and base of all applications. Because of the enormous size and complexity of OS, the accuracy is not easy to describe and illustrate. Despite intensive testing, the bugs in OS do occur over time, which can be seen from the fact that the current mainstream commercial OSs continually release the update patches. How to ensure the correctness of OS is the direction of industry and academia efforts. Using formal method for OS development is an effective way of achieving this aim.

Formal methods have rigorous mathematic semantics to do formal description and verification, so they can guarantee the correctness of the software program. A literature search for publications describing the use of formal methods, in the context of OS, identifies the following work in this area.

To start with, [1] and [2] presented formal models and refinement of operating system kernels using Z. [3] proposed further a simple and correct specification of an OS kernel in Z which simplified the understanding and verification of operating system components. Besides, [4] applied Event B to develop a formal model of the API of the L4 microkernel. The goal was to evaluate the possibility to model such software with formal techniques. By contrast, [5] and [6] focus on correctness verification of OS. Specifically, [5] introduced an OS state automaton model as a link between the system design and verification, and described the correctness specifications of the system based on this model. Then it implemented the verified trusted operating system as a prototype, to illustrate the method of consistency verification of system design and safety requirements with formalized theorem prover Isabelle/HOL. As to [6], there presented a detailed coverage of the comprehensive formal verification of the seL4 microkernel, from its initial functional correctness proof to more recent results, which extended the assurance argument up to higher-level security properties and down to the binary level of its implementation. Different from existing research, this paper employs B method to design a formal model of OS.

B method covers software development process from abstract specification to an implementation through successive refinement steps [7]. Taking the precise mathematics semantics as the foundation, B method supports rigorous development process. With the top-down formal specifications and related proofs, the developers may find and get rid of many design and implementation errors in the early development stages so the resulting software can be more coherent and reliable [8]. Furthermore, there exist industrial software tools that support the B method. By the use of tools supporting the B method, we are able to formally verify and refine systems with higher speed and accuracy. These are main advantages of B over other formal methods such as using Z or VDM.

Based on the advantages of B method, it is worth researching to apply B method for the rigorous development of OS. Therefore, this paper takes Micro-Controller Operating Systems-II (μC/OS-II) as a reference and uses B method to design a formal model of OS called fmC/OS. The paper shows the fmC/OS development process from abstract specification to an implementation.

The remainder of this paper is organized as follows. At first, section 2 presents the μC/OS-II system. Then section 3 details the Abstract Model. Section 4, afterwards, describes the Concrete Model. And finally section 5 summarizes and concludes the paper.

2. MICRO-CONTROLLER OPERATING SYSTEMS-II PRESENTATION

Micro-Controller Operating Systems-II (μC/OS-II) is a real-time operating system designed by embedded software

1874-110X/15

developer, Jean J. Labrosse in 1998. It is priority-based preemptive real-time for microprocessors, written mainly in the C programming language [9]. µC/OS-II is applied in the following embedded systems: Avionics, Medical equipment/devices, Data communications equipment, White goods, Mobile phones, PDAs, Industrial controls, Consumer electronics and Automotive.

Main function units of µC/OS-II include task scheduler, task management, inter-task synchronization and communication, and memory management. µC/OS-II is a multitasking operating system. Each task is an infinite loop and can be in any one of the following five states: Dormant, Ready, Running, Waiting and Interrupted. Additionally it can manage up to 64 tasks. Task priorities can range from 0 (highest priority) to a maximum of 63 (lowest possible priority). Each task runs at a different priority, and thinks that it owns the CPU. Lower priority tasks can be preempted by higher priority tasks at any time. The system user of µC/OS-II is able to control the tasks by using features as follows: Task Feature, Task Creation, Task Stack & Stack Checking, Task Deletion, Change a Task's Priority, Suspend and Resume a Task, and Get Information about a Task. In order to avoid fragmentation, µC/OS-II allows your application to obtain fixed-sized memory blocks from a partition made of a contiguous memory area. All memory blocks are in the same size and the partition contains an integral number of blocks. Inter-task or inter-process communication in µC/OS-II takes place using Semaphores, message mailbox, message queues, tasks and Interrupt service routines (ISR). They can interact with each other when a task or an ISR signals a task through a kernel object called an Event Control Block (ECB). The signal is considered to be an event.

3. ABSTRACT MODEL

We take µC/OS-II as a chief reference and apply B method to develop a formal model of Operating System named fmC/OS. The abstract model architecture is based on the functional breakdown provided by the informal description. Hence, we divide the Operating System into eight subsystems. Afterwards, we use "INCLUDES" to construct the machine, OS, which is the formal model of µC/OS-II. Table **1** lists names and functions of these sub-systems' models.

Table 1. Names and functions of the sub-systems.

Names	Functions
Task	Task management
OS_Sched	Task scheduler
Event_Control_Blocks	Management of Event Control Blocks
Semaphore	Semaphore management
Mutex	Mutex semaphore management
Mailbox	Mailbox management
Mailqueue	Mail queue management
Memory	Memory management

Due to limited space, only the machine, OS_Sched, of eight sub-systems is expressed in this paper and the description of specifications uses Atelier B4.0's syntax and notation.

The machine, OS_Sched, is in charge of the ready queue and all the waiting queues. µC/OS-II is a priority-based preemptive real-time operating system so the ready queue and all the waiting queues sort by priority of tasks. That is, tasks with high priority are placed in the front of a queue while those with low priority are at the back. It has to be noted that operations of ready queue are rather similar to those of waiting queues so we introduce a machine named Queue to describe relevant operations of sorted queues in order to improve reuseability.

The machine, Queue, can provide operations including: inserting a task into a sorted queue (Insert_element), reading the first task from the queue (Nextfromqueue) and deleting the first task of the queue (Delete_element). Fig. (**1**) shows the specification of Insert_element.

```
newqueue<--Insert_element(queue,did)=
    PRE
        did:tid&
        queue:iseq(tid)&
        did/:ran(queue)
THEN
    SELECT queue=[] THEN
        newqueue:=[did]
    WHEN prio(did)>prio(first(queue)) THEN
        newqueue:=did->queue
    WHEN prio(did)<prio(last(queue)) THEN
        newqueue:=queue<-did
    ELSE
        ANY s1,s2
        WHERE
            s1:iseq(tid)&
            s2:iseq(tid)&
            queue=s1^s2&
            s1/=[]&
            s2/=[]&
            prio(last(s1))>prio(did)&
            prio(did)>prio(first(s2))
        THEN
            newqueue:=s1^[did]^s2
        END
    END
END
```

Fig. (1). Operation insert_element of the machine queue.

The machine, OS_Sched, needs to see sets, constants and variables of the machine, Task, the set of the machine, Basic Status, and the set of the machine, Event_Control_Blocks. To switch tasks, OS_Sched needs to call the operation, osctxsw, of the machine, OS_Task_SW. It is why OS_Sched

includes OS_Task_SW. It is a common operation to delete a task from one queue and then add the task into the other queue in OS_Sched so OS_Sched also includes two copies of the machine, Queue.

OS_Sched defines four variables: readyqueue, waitqueue, current and eventwaitqueue which represent ready queue, commonly waiting queue, current task and queue waiting for certain event respectively.

```
MACHINE
    OS_Sched
SEES
    Task,Basic_Status,Event_Control_Blocks
INCLUDES
    OS_Task_SW,xx.Queue,yy.Queue
VARIABLES
    readyqueue,waitqueue,current,eventwaitqueue
INVARIANT
    readyqueue:iseq(Tid)&
    readyqueue/=[]&
    waitqueue:iseq(Tid)&
    eventwaitqueue:Event_Block+->iseq(Tid)&
    current:ran(readyqueue)&
    idletask=last(readyqueue)&
    ran(readyqueue)/\ran(waitqueue)={}&
    !(xx,yy).(xx:dom(eventwaitqueue)&yy:dom(eventwaitqueue)
&xx/=yy=>
    ran(eventwaitqueue(xx))/\ran(eventwaitqueue(yy))={})&
    !xx.(xx:dom(eventwaitqueue)=>ran(eventwaitqueue(xx))/\ran
(readyqueue)={})&
    !xx.(xx:dom(eventwaitqueue)=>ran(eventwaitqueue(xx))/\ran
(waitqueue)={})
INITIALISATION
    readyqueue:=[idletask]||waitqueue:=[]||
    current :=idletask||eventwaitqueue:={}
OPERATIONS
    dispatch=
        PRE
            readyqueue/=[]
        THEN
            IF current/:ran(readyqueue) or
prio(current)<prio(first(readyqueue))
            THEN
                current:=first(readyqueue)||
                osctxsw
            END
        END;
  ...
```

Fig. (2). The machine OS_Sched.

As Fig. (**2**) indicates, the machine, OS_Sched, has to satisfy many constraints which are showed in its invariant. The invariant not only restricts types of variables but also describes the relationship between variables and constants. Specifically, the type of ready queue is injective sequence of the set, Tid. It ensures that the same task will never appear twice in the ready queue. So is commonly waiting queue. Then, ready queue cannot be null. That is, there is always one idle task in the queue. The idle task, due to its lowest priority, is the last one of the ready queue. Besides, current task also comes from the ready queue. One task, furthermore, cannot appear simultaneously in both ready queue and waiting queue. Similarly, the same task will never exist in any two different waiting queues at the same time, either.

The machine, OS_Sched, has thirteen operations of which only the operation, dispatch, is presented in Fig. (**2**).

In the similar way to develop OS_Sched, we develop other seven subsystems which are described by seven machines respectively as follows: Task, Memory, Event_Control_Blocks, Mailqueue, Semaphore, Mutex and Mailbox. We can, further, construct a big specification named OS. Fig. (**3**) illustrates part of OS.

```
MACHINE
    OS
SEES
    Message,Basic_Status
INCLUDES
    Task(0..63),OS_Sched,Memory(1..1024),
    Event_Control_Blocks,Mailqueue, Mutex,Mailbox,
Semaphore
PROMOTES
    Createidletask,TaskQueryStatus, MemPut, MemGet,
    MemCreate,MemDelete, MemQuery, TaskQueryPri
```

Fig. (3). Part of the machine OS.

The clause, PROMOTES, as Fig. (**3**) shows, lists operations of the machine, Task: Createidletask, TaskQueryStatus and TaskQueryPri and operations of Memory: MemCreate, MemDelete, MemGet, MemPut and MemQuery. All the operations are promoted as operations of OS. Besides these operations mentioned above, the machine, OS, can also provide another thirty-four operations of which definitions are not stated here.

We can find from the clause, includes, that parameters of Task and Memory are assigned. Values of these parameters can alter according to actual conditions of system. This approach, on the one hand, improves flexibility of design to the maximum extent and postpones detail decision-making as long as possible. It facilitates us to handle different kinds of problems arising in the process of development such as changes of requirements. On the other hand, the method not only lowers complexity of software but also enhances modifiability and extendibility of software.

Table 2. Automatic proof rate of machines.

Component	Proof Obligations	Proved	Unproved	Automatic Proof Rate %
AddressData	2	2	0	100
Basic_Status	0	0	0	100
Event_Control_Blocks	8	8	0	100
Mailbox	6	6	0	100
Mailqueue	87	71	16	81.6
Memory	46	40	6	86.9
Message	0	0	0	100
Mutex	6	6	0	100
OBJECT	6	6	0	100
OS	6	2	4	33.3
OS_Sched	172	74	98	43.0
OS_Task_SW	0	0	0	100
Query	0	0	0	100
Queue	0	0	0	100
Sched_Query	0	0	0	100
Semaphore	6	5	1	83.3
Setobject	2	2	0	100
Task	62	61	1	98.3
Total	409	283	126	69.2

B method can generate proof obligations in every stage of the development and prove the obligations by rigorous mathematical proof. In the initial stage of specifications, the proof obligations are invariant theorems and are used to check whether the operations of the specification satisfy the invariant. In the refinement stage, the proof obligations are refinement relationship theorems and verify that the refinement is consistent with the original abstract model. Errors may be found during the process of the proof, like misunderstanding, false demand and incongruous system design. With related proof, the developers may find and avoid many design and implementation errors so the resulting software can be more coherent and reliable.

The paper applies Atelier B to develop the system. The Atelier B, as computer software developed by ClearSy company, can be used to support the B method. The Atelier B has numerous tools such as a powerful editor with the ability to warn the user in the case of mistyping or even potential typing errors, automatic proof obligation generator, automatic prover, and an interactive prover. The automatic prover of the Atelier B is very effective. As Table **2** indicates, the proportion of proof obligations discharged by auto-proof varies by machines and overall is about 69.2 % (283 out of 409). When the abstract model is fully proved, we are sure that the abstract model complies with invariant theorems.

4. CONCRETE MODEL

The concrete Model phase consists in completing the Abstract Model to get to a completely implementable B project. The only input of this phase is the abstract model and the goal is to implement it completely through refinement and importation breakdown. When the concrete model is fully proved, we are sure that the concrete model complies with the abstract model.

The paper takes implementation of the machine, OS, as an example. OS_i, as showed by Fig. (**4**), is implemented by importing machines: Task, OS_Sched, Memory, Mailqueue, Semaphore, Mutex, Event_Control_Blocks, Mailbox and Query and calling operations of these machines.

The operations of the machine, OS_i, are deterministic and very close to the structure of specific programming language. They can, thus, correspond to statements of the classical programming language like the order of statement, conditional statement and procedure call. It establishes a good foundation for translating the model into executable code.

```
IMPLEMENTATION
    OS_i
REFINES
    OS
SEES
    Message,Basic_Status
IMPORTS
    Task(0..63),OS_Sched,Memory (1..1024),
    Event_Control_Blocks,Mailqueue,
    Semaphore,Mutex,Mailbox,Query
PROMOTES
    Createidletask,TaskQueryStatus, MemGet, MemPut,
    TaskQueryPri,MemCreate, MemRelease, MemQuery
OPERATIONS
    OSTaskChangePrio(did,newpri)=
        VAR bb1,bb2,bb3,eid IN
          bb1<--isnot_idletask(did);
          IF bb1=TRUE
          THEN
            ChangePri(did,newpri);
            bb2<--inreadyqueue(did);
            IF bb2=TRUE
            THEN
              delete_from_readyqueue(did);
              add_to_readyqueue(did);
              dispatch
            ELSE
              bb3<--inwaitqueue(did);
              IF bb3=TRUE
              THEN
                delete_from_waitqueue(did);
                add_to_waitqueue(did)
              ELSE
                eid<--ineventwaitqueue(did);
                delete_from_eventwaitqueue(eid,did);
                add_to_eventwaitqueue(eid,did)
              END
            END
        END
    END
…
```

Fig. (4). The implementation OS_i.

CONCLUSION

The paper uses B method to design a formal model of OS and shows the fmC/OS development process from abstract specification to an implementation. We have obtained abstract model and properties of the system and for each operation its prerequisites as well as the invariant must be satisfied. Every verification stage of B method is useful and leads to error detection: analysis of software document specification, type checking, inspections, proof of abstract model safety properties, refinement proof of correct implementation. Therefore, Operating System based on B method can enhance accuracy and security of the system.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflict of interest.

ACKNOWLEDGEMENTS

We gratefully acknowledge support by the Foundation and Forefront Technology Research Project of Henan Province (122300410062) and the Natural Science Research Project of Education Department of Henan Province (12A520007).

REFERENCES

[1] Iain D. Craig, Formal models of operating system Kernels, Springer, 2007.

[2] Iain D. Craig, Formal refinement for operating system Kernels, Springer, 2007.

[3] Luciano Barreto, Aline Andrade, Adolfo Duran, Caique Lima and Ademilson Lima, "Abstract specification and formalization of an operating system kernel in Z," Operating Systems Review, vol. 45(1), pp. 156-160, Jan. 2011.

[4] Sarah Hoffmann, Germain Haugou, Sophie Gabriele and Lilian Burdy, "The B-Method for the Construction of Microkernel-Based Systems," 7th International Conference of B Users, Besançon, 2007, pp. 257-259.

[5] Zhenjiang Qian, Hao Huang, and Fangmin Song, "VTOS: Research on Methodology of 'Light-Weight' Formal Design and Verification for Microkernel OS," Information and Communications Security - 15th International Conference, Beijing, 2013, pp. 17-32.

[6] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser, "Comprehensive formal verification of an OS microkernel," ACM Transactions on Computer Systems, vol. 32, pp. 2:1-2:70, Feb. 2014.

[7] J.-R. Abrial, The B-Book: Assigning Program to Meanings, CUP, 1996.

[8] Jean-Raymond Abrial, "Formal methods in industry: achievements, problems, future," 28th International Conference on Software Engineering,, Shanghai, 2006, pp. 761-768.

[9] Jean J. Labrosse, MicroC OS II: The Real Time Kernel (2nd Revised edition), CMP Books, 2002.

Received: June 10, 2015 Revised: July 29, 2015 Accepted: August 15, 2015