A New OS Structure for Simplifying Understanding of Operating System Behavior

Toshihiro Yamauchi, Akira Kinoshita, Taisuke Kawahara and Hideo Taniguchi
Graduate School of Natural Science and Technology, Okayama University
3-1-1 Tsushima-naka, Kita-ku, Okayama, 700-8530 Japan
E-mail: yamauchi,tani@cs.okayama-u.ac.jp

Abstract

It is difficult to understand the processing flow of complicated software such as operating systems (OSs). Thus, a mechanism that can collect and analyze behavioral information in order to comprehend the behavior of OS is necessary. Although several collection mechanisms have been developed, their OS structures were not designed to collect OS behavior. In this paper, we describe an OS structure that simplifies comprehension of OS behavior and the implementation of it on ***Tender*** OS. We also describe a mechanism for the visualization of OS behavior. Finally, we investigate the cost of introducing our proposed comprehension mechanism and the overhead and efficiency of the proposed mechanism.
Keywords : Understanding OS behavior, Visualization, Operating system, OS structure

1 Introduction

Software size has been increasing and many functions are deployed in recent software. If the size of software increases, analysis of software behavior at runtime becomes increasingly difficult. In addition, it is difficult to understand the processing flow of complicated software such as operating systems (OSs). In order to comprehend the internal behavior of an OS, the series of program execution information in an OS can be utilized. Furthermore, to support an understanding of OS behavior, it is necessary to visualize the program execution information recorded in an OS.

There are some research efforts that are geared towards collecting program execution information and supporting the understanding of software behavior [1-4]. Their purpose is to support OS kernel development. Because they have to collect various pieces of information, they implement many hook functions in an OS. However, it is difficult to add a collection function at the target function in an OS.

To the best of our knowledge, few OSs are primarily designed with comprehension of OS internal program behavior in mind. In this paper, we first outline the drawbacks of existing methods for comprehending OS behavior, and then we propose an OS structure that is designed to aid the understanding of OS behavior, and describe its implementation on the ***Tender*** operating system. Our proposed OS structure introduces a unified interface to call OS functions in the OS kernel. Every function must call a unified interface controller to call other functions. Second, OS programs are divided into program modules depending on the operation, target resource, and operation type. This mechanism can help a user understand OS structure and functions. Third, this mechanism can collect information on calling functions in the OS kernel. To record this information in a unified interface controller, all of the OS behavior can be understood in our proposed OS structure. Finally, it looks at the cost, overhead and efficiency of our proposed OS structure.

2 Drawbacks of existing kernel trace functions

LKST [1], Systemtap [2], and DProbe [3] deploy some hooks in the OS kernel. When a function deployed on the hook is executed, the hook function for collecting execution information is invoked. DTrace [4] provides a function that modifies the OS kernel and user processes to record additional data that the user specifies at locations of interest. In these research efforts, a user can not only use default hooks, but also additional hooks that may interest the user. Consequently, a user can get detailed information about the kernel at runtime.

However, in order to insert the additional hooks, a user needs to have knowledge of the OS. In addition, it is difficult to insert appropriate hooks into the OS kernel. Furthermore, if kernel trace functions require patches for deployment, the modification of kernel version up is not easy for the developer of kernel trace functions.

Oprofile is a research effort for OS profiling [5]. It can get statistical information of an OS by measuring the frequency of program execution. However, as this information is only statistical, Oprofile cannot be used to analyze program behavior. LSMPMON [6] records the processing time at the each hook point of LSM and the calling count. However, it can only collect program execution information of secure OS based on LSM.

3 OS structure for simplifying understanding of operating system behavior

3.1 Requirements

The requirements of an OS structure for simplifying comprehension of OS behavior are as follows:

Requirement 1: Modularization

Programs in the kernel of the OS should be classified into program modules based on the kind of resource and the type of operation. When each program is labeled in terms of kind and type, an information collection module can understand the flow of processing and the contents of processing in the kernel of the OS.

Requirement 2: Unified calling interface for modules

A unified interface to call OS functions should be introduced in the OS kernel. All function calls should then pass through the unified interface to call other OS functions. Thus, the unified interface controller can collect information associated with all function calls. In this OS structure, only one collection module in the unified interface controller is required for collecting OS behavior information.

3.2 Design of program module and a unified interface

In order to modularize OS functions, a program is divided into meaningful functional units. Every program that operates the resources that the OS manages as a unit of a function is modularized. Furthermore, a program that operates the same resource is divided into five operations: OPEN, CLOSE, READ, WRITE, and CONTROL. Each operation of each resource corresponds to a unit of a module. The call interface of all the modules is unified in our proposed structure. For example, the OPEN operation is called using the open_rsc() function. The first argument of open_rsc() is a kind of resource. By specifying the kind of resource and the function name, a program module can be called.

In order to understand the call relationship between modules, the following characteristics are required of each module:

1. Call of a module and return from a module should be detectable.
2. Each module should be distinguishable.

The call of a module is controllable by fulfilling the above characteristics. The call of all the modules is systematically manageable by unifying control of the call of each module.

3.3 Structure of OS behavior visualization mechanism

The visualization processing is divided into four parts:

1. Collection of visualization information (collection part)
2. Visualization information data transfer (transfer part)
3. Analysis of operation based on visualization information (analysis part)
4. Visualization of analytical data (visualization part)

Collection part collects program execution information at a unified interface, and records the collected information. Because the collection part should continuously monitor processing, which is the target of information collection, it is implemented inside the OS. The analysis and visualization parts perform data processing and processing of visualization, respectively. Analysis part inspects visualization information; classifies according this information to the analysis part for every identifier, kind of resource, and kind of operation; and totals the number of times a call is made and the processing time of each resource processing. In the transfer part, the information collected in the collection part is written out to a disk and saved. This saved information is used in the analysis and visualization parts. Visualization part outputs SVG picture in order to support the comprehension of the OS operation. The details of these four parts are written in paper [7].

4 Evaluation of the proposed OS structure

4.1 Implementation on *Tender* operating system

In order to satisfy requirement 1, management in Section 3. was performed. In ***Tender***, OS resources can exist independently. The objects to be controlled and managed by ***Tender*** are known as “resources.” The programs that use each resource are separated from each other. The program modules consist of five program components, namely, “open,” “close,” “read,” “write,” and “control.” Resources have identifiers and names for operations. The interface for the operation of resources is a unified interface called the Resource Interface Controller (RIC). Programs that use resources are called through the RIC. Each program that manages resources has to call program components through the RIC. Bypassing the RIC is prohibited in the ***Tender*** kernel. Therefore, if the

Table 1: Example of the collected information at RIC

id	src_id	dest_rid	rsc_kind	operation	ret_val	flowid	call_k_time	ret_k_time
4686	-1	d0140	VMEM	OPEN	140	a0008	df894b764	df899c735
4687	4686	b068b	VREG	OPEN	68b	a0008	df894d4cf	df89593cb
4688	4687	2284e	PMEM	OPEN	284e	a0008	df8958de0	df8958f89
4689	4686	30137	VKM	OPEN	137	a0008	df895a2d2	df8971a2c
4690	4689	b068b	VREG	CONTROL	1000	a0008	df895ac1a	df895ada9
4691	4689	d0001	VMEM	CONTROL	0	a0008	df895af19	df89718e9
4692	4691	b068b	VREG	CONTROL	1000	a0008	df895b1fa	df895b38e
4693	-1	80000	EXEC	CONTROL	80009	ef0000	df8962618	df8968b2d
4694	4691	b068b	VREG	CONTROL	284e000	a0008	df897137f	df897151b

information on the call to a program section is recorded, the call relationship function of the OS can be understood. Table 1 shows the example of the collected information at RIC.

In ***Tender***, original functions are implemented. A new type of execution resource is implemented. It can control the maximum CPU usage such that the abuse of CPU resources can be prevented [8]. In addition, by preserving and recycling process resources, ***Tender*** can realize a reduction in the cost of process creation and termination [9].

Below, the results evaluated about the application to introduction cost, a performance overhead, and a performance improvement and grasp of OS operation are described.

4.2 Cost of introduction

As comparison for information gathering, the number of information gathering parts for LKST, DTrace, and ***Tender*** were compared.

The number of information gathering parts for LKST is about 100 places in a default setup at the time of introduction. In DTrace, it is 30,000 or more places, which is scalable according to the system. Since there are many information collection parts, it is difficult to understand all the parts where information collection is performed in these systems.

On the other hand, in ***Tender***, since resource processing is managed in a unified manner by RIC, information can be collected at one place and understanding of the information collection part is easy. Moreover, since a call for resources always passes through RIC, when adding new OS modules, there is no need for modification to add information collection for the new modules.

4.3 Performance overhead

The overhead of the proposed mechanism in ***Tender*** was measured for each resource processing. Specifically, the execution time in the case where information collection processing is effective and invalid was measured for 66 system calls in ***Tender***. A 1 GHz Pentium III computer was used for the measurement. The increased time per resource processing call was about $0.5 \mu s$ on average, and $1 \mu s$ at the maximum. The increase percentages were 0.7% on average and 66.4% at the maximum.

4.4 A case of performance improvement

The Apache Web server was run on ***Tender*** and the performance analysis was performed on ***Tender*** for the performance improvement. What the processing time of each resource in the descending order is shown in Figure 1. Figure 1 shows that resource processing that processing time is the largest is creation

	flowid	rsc	ope	num	avetime	totaltime	variance
1	SYSCALL	VREGION	OPEN	65	1434	93212	762578
2	INTR	EXEC	CONTROL	39	1410	55009	6145747
3	SYSCALL	PROC	WRITE	12	4136	49639	1596882
4	SYSCALL	PROG	READ	21	1204	25291	1271088
5	SYSCALL	VMEMORY	CONTROL	352	38	13668	1887
6	SYSCALL	VUM	OPEN	168	66	11209	3028
7	SYSCALL	VREGION	CLOSE	62	90	5606	726
8	SYSCALL	VREGION	CONTROL	3997	0	3663	104
9	SYSCALL	PLATE	CONTROL	6	459	2756	8150
10	UNKNOWN	VMEMORY	CONTROL	12	196	2357	26717

[μ sec.]

Figure 1: List of analyzed information of OS behavior using Apache Web server

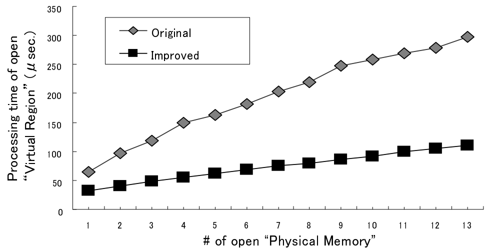

Figure 2: Processing time of creation of virtual region

of a virtual region. Then, the creation processing of a virtual region was investigated using visualization information. The processing time of a physical memory resource creation is about 1 ms. However, the processing of a physical memory resource creation includes an overhead which is about 50 ms to 100 ms. From this, it is thought that the processing of a physical memory resource creation involve large overhead before or after the processing.

Since a resource name was assigned to each resource in resource creation processing, OS checks duplication of a resource name before the resource creation. The investigation by the analysis showed that this duplication check had taken big time.

By generating a resource name from a resource identifier and the page number of created physical page, OS avoid the duplication of the resource name collision. The relation between the processing time of open "virtual region" and a number of created physical memory resource is shown in Figure 2. Figure 2 shows that this modification can reduce processing time more than 50%.

4.5 A case of understanding OS behavior

We show process scheduling and an exception as an example of understanding OS behavior by the proposed method. In this experiment, when three processes are run, the proposed system recorded the information of OS behavior, and visualized them. The visualized information is shown in Figure 3. PROC means a process, SYSCALL means a system-call processing, PREEMPTION and TIMESLICE mean the time of preemption and time slice processing, and EX means an exception.

In ***Tender***, process execution speed regulating function using "execution" resource is implemented. The process a000c, the process a000d, and the process a000e are assigned the processor processing time 10%, 30%, and 50%, respectively. Figure 4 shows that the ratio of the assigned processor time is correspond to 10%, 30%, and 50%. Figure 4 shows the part of an exception in figure 3. Figure 4 shows that an exception ee000e is recorded and visualized in our method. Therefore, we can analyze events before and after the exception by using the

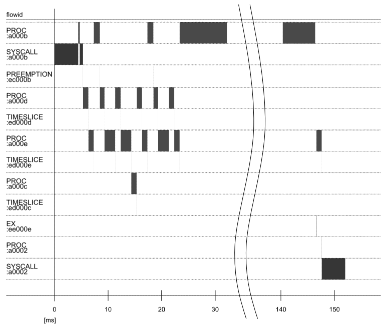

Figure 3: Visualized OS behavior

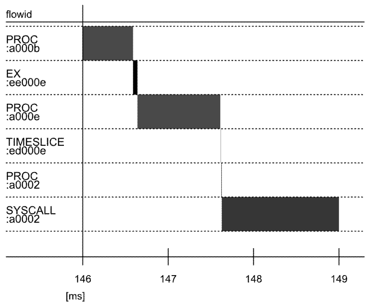

Figure 4: Expanded figure of an exception

visualized information.

5 Conclusion

This paper proposes an OS structure that simplifies comprehension of OS behavior. A unified interface is introduced in this structure for understanding all calls made by program modules. In addition, its implementation in the ***Tender*** operating system is described. This OS structure consists of four parts: collection, transfer, analysis, and visualization parts. Evaluation results show that the cost of introduction and overhead of the proposed OS structure are reasonable, and show the efficiency of the proposed structure in two cases.

References

[1] Hitachi Ltd., Fujitsu Ltd., "Linux Kernel State Tracer," Online publishing, http://lkst.sourceforge.net/ , 2001.

[2] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, Brad Chen, "Locating System Problems Using Dynamic Instrumentation," In Proceedings of the Linux Symposium, Vol. 2, pp.49-64, 2005.

[3] Richard Moore, "A Universal Dynamic Trace for Linux and other Operating Systems," Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, pp.297-308, 2001.

[4] B. M. Cantrill, M. W. Shapiro, A. H. Leventhal, "Dynamic Instrumentation of Production Systems," USENIX Annual Technical Conference, pp.15-28, 2004.

[5] J. Levon and P. Elie, "Oprofile: A system profiler for linux," http://oprofile.sf.net/ , 2004.

[6] T. Yamauchi, K. Yamamoto, "LSMPMON: Performance Evaluation Mechanism of LSM-based Secure OS," International Journal of Security and Its Applications (IJSIA), Vol.6, No.2, pp.81-89, 2012.

[7] T. Yamauchi, A. Kinoshita, T. Kawahara, H. Taniguchi, "Design of an OS Architecture that Simplifies Understanding of Operating System Behavior," Proc. International Conference on Information Technology and Computer Science (ITCS 2012), pp. 51-58, 2012.

[8] T. Tabata, S. Hakomori, K. Yokoyama, H. Taniguchi, "A CPU Usage Control Mechanism for Processes with Execution Resource for Mitigating CPU DoS Attack," International Journal of Smart Home, Vol. 1, No. 2, pp.109-128, 2007.

[9] T. Tabata, H. Taniguchi, "An Improved Recyclable Resource Management Method for Fast Process Creation and Reduced Memory Consumption," International Journal of Hybrid Information Technology (IJHIT), Vol.1, No.1, pp.31-44, 2008.