International Journal of Smart Home
Vol. 1, No. 2, July, 2007

A CPU Usage Control Mechanism for Processes with Execution Resource for Mitigating CPU DoS Attack

Toshihiro Tabata
Graduate School of Natural Science and Technology, Okayama University
tabata@cs.okayama-u.ac.jp

Satoshi Hakomori
Graduate School of Natural Science and Technology, Okayama University
NTT DATA Corporation
hakomoris@nttdata.co.jp

Kazutoshi Yokoyama
NTT DATA Corporation
yokoyamakz@nttdata.co.jp

Hideo Taniguchi
Graduate School of Natural Science and Technology, Okayama University
tani@cs.okayama-u.ac.jp

Abstract

In a ubiquitous environment, the hardware resources are limited; thus, an appropriate resource management mechanism is required for guaranteeing its processing activity. However, most operating systems (OSs) lack an access control mechanism for CPU resources to guarantee satisfactory processing and to safeguard the system from malicious attacks that affect the CPU resources, resulting in denial of service (DoS). Access control is not intended for CPU resources, which are important for the execution of a program. As a result, OSs cannot control the usage ratio of CPU resources. In this paper, we propose an access control model for CPU resources based on an execution resource. The proposed model can control the usage ratio of CPU resources appropriately for each user and each program domain. This execution resource can be applied to mitigate DoS attacks. In order to evaluate the effectiveness of the proposed method, we describe the results of a basic performance experiment and a DoS simulation experiment employing the Apache web server. From the results, we show that the proposed method can mitigate DoS attacks and does not have bad effects upon the performance of a target service.

1. Introduction

With the spread of computers and the Internet, the number of reports on software attacks is increasing with time. The widespread use of automated attack tools and attacks against Internet-connected systems are common and therefore the number of attack incidents is also increasing [1]. In addition, denial-of-service (DoS) attacks are one of the serious problems in computer systems. Therefore, various defense mechanisms against such attacks have been actively studied.

A firewall blocks packets from incoming connections, but cannot evaluate the contents of “legitimate” packets. Besides, intrusion detection systems (IDS) [2], [3], [4] observe program executions and detect malicious behaviors of the program. The incidence of false negatives and false positives is not zero on IDS. Therefore the combination of a firewall and IDS is not sufficient to prevent attackers from attacking computers.

In a ubiquitous environment, the hardware resources are limited and the processing must be performed with the limited hardware resources. Thus, an appropriate resource management mechanism is required for guaranteeing the processing ability of the environment. In particular, the CPU power in a ubiquitous environment is lower than that in other environments such as server systems and cluster systems. Therefore, the CPU time must be appropriately assigned to processes on the basis of their priority. However, most operating systems (OSs) do not have an access control mechanism for CPU resources to guarantee satisfactory processing and to safeguard from malicious attacks that affect the CPU resources, resulting in DoS.

Access control is not intended for CPU resources, which are important resources for the execution of a program. As a result, such OSs cannot control the usage ratio of CPU resources. For example, a secure OS [5] employs mandatory access control (MAC) and role-based access control (RBAC) [6]. Even if the authority to control the usage ratio is assumed, the secure OS renders the range of influence a minimum by implementing the least privilege policy. However, access control is not intended for CPU resources. As a result, a secure OS is not able to prevent attackers from carrying out DoS attacks, which affects CPU resources. In general OSs, attacks can limit only the maximum CPU time for each process and not the proportion of CPU time among the processes.

In an earlier study, we proposed an execution resource for regulating and guaranteeing the performance of processes [7], [8]. In this paper, we propose a new type of execution resource that controls the maximum extent of CPU usage for preventing abuse of CPU resources. In addition, we propose an access control model for CPU resources based on an execution resource. Each execution has a degree of CPU usage. The CPU time for the execution of a process is assigned by a scheduler according to the degree of CPU usage.

The proposed model can control the usage ratio of CPU resources appropriately in each program domain. The proposed execution resource enables system administrators to control the maximum extent of CPU usage for each user and each program domain because the access control of CPU usage is performed in a similar manner as an object such as a file. In other words, this execution resource can be applied to mitigate DoS attacks. Finally, we describe the results of experiments employing the Apache web server.

2. Related Work

Most defense techniques against Internet-originated DoS attacks have targeted the transport and network layers of the TCP/IP protocol stack [9]. Our research focuses on the access control mechanism of and the rate limiting technique for CPU resources.

In the past decade, resource accounting techniques and resource protection techniques for defending against DoS attacks have been proposed, and these techniques have been successfully utilized to counter DoS attacks. The Scout operating system has an accounting mechanism for all the resources employed by each I/O path [10]. Scout also has a separate protection domain for each path. The present research focuses on the I/O paths and not on the access control of CPU resources.

The resource control mechanism is implemented by the resource control lists (RCL) in Linux RK [11]. This mechanism involves the resource reserve function and rate limiting function for controlling the CPU time, and this mechanism controls a process as a unit of RCL and enforces the RCL rules to it. Thus, this mechanism cannot control the CPU time of users using the rate limiting function.

Resource containers [12] have been proposed, and they can be used to account for and limit the usage of kernel memory. This container mechanism supports a multilevel scheduling policy; however, it only supports fixed-share scheduling and regular time-shared scheduling.

Execution resources with upper bounds are classified under resource accounting techniques [13]. The execution resource can control the maximum extent of CPU usage of programs for preventing abuse of CPU resources. The policy of rate limiting can be enforced for a CPU resource by using an access control mechanism for the execution resource. In addition, the proposed access control model can be applied to general OSs and secure OSs.

3. Execution Resource

In this section, we explain the concept of execution resource on the basis of the presentation in previous papers [7], [8].

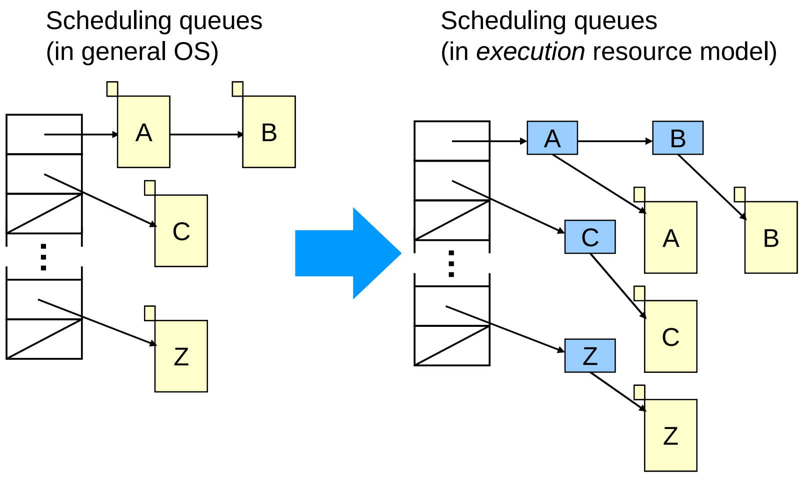

Figure 1. Change of scheduling queues

3.1. Overview

A process may be described as a unit of program execution in an existing OS, and it has a degree of CPU usage. For example, a priority is associated with each process in UNIX. We have separated the degree of CPU usage from a process. The degree of CPU usage has been named execution resource. Therefore, only the execution resource involves the degree of CPU usage, and a process does not have a degree of CPU usage. Scheduling queues are shown in Figure 1. Prior to the introduction of the execution resources, processes are listed on a linked list on the basis of their priority. After the introduction of the execution resources, it is these execution resources that are maintained on the linked list on the basis of their priority. Processes are then linked to executions. A process can be executed by linking it to an execution.

Figure 2 shows the relation between executions and processes. The execution manager points to an execution with the highest degree of CPU usage. All processes need to be linked to executions to be assigned a CPU time. The execution manager selects a process from the scheduling queues. When the state of a process is READY, it is linked to the execution with the highest priority. The amount of CPU time that is assigned to a process is proportional to the total amount of CPU usage time required for the executions linked to the process.

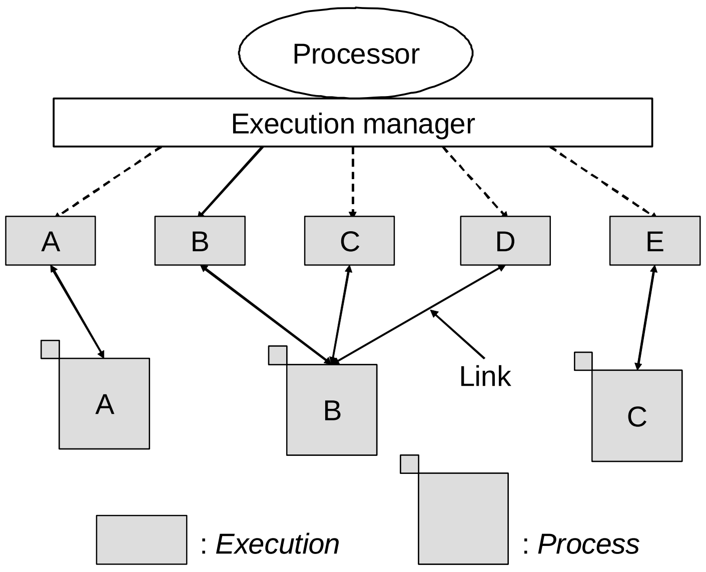

Figure 2. Relation between executions and processes

3.2. Types of Execution Resources

There are two types of execution resources. One is execution with performance and the other is execution with priority.

Execution with performance: Execution with performance includes a degree of CPU usage that indicates the proportion of bare processor performance. The bare processor performance can be defined as 100%. When a process is linked to an execution with n% performance, the assigned CPU time is n% of the bare processor performance. We named a unit of CPU usage as a “time slot.” We termed a group of time slots as a “time block.” Figure 3 shows the relation between time slots and a time block. An execution in which the degree of CPU usage is n% is assigned n% of the time slots in a time block.

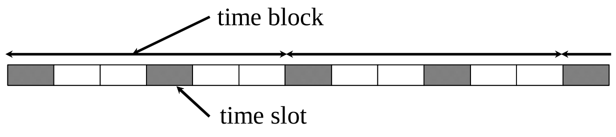

Figure 3. Time slots and a time block

Execution with priority: Execution with priority includes the degree of performance that indicates the priority. The execution manager assigns the execution with priority that has the highest priority to the processor. However, execution with performance takes precedence over execution with priority because the former is guaranteed an assigned CPU time.

3.3. Hierarchical Execution Tree

The structure of a process group is represented as a tree structure of executions because the relation between a process group and its processes is represented as a parent and a child. Figure 4 shows the relationships between a process group and executions. The node of an execution tree is called “directory execution” and it represents the degree of CPU usage for a process group. A leaf is called “leaf execution”; every leaf execution is linked to a process.

The total assigned CPU time for leaf executions equals the assigned CPU time for the parent directory execution. The degree of CPU usage for leaf executions indicates the priority or a ratio (%) to assigned CPU time of parent directory execution. In the leaf execution, the ratio corresponds to the point where the parent directory execution is defined as 100%. The depth of an execution tree is greater than one. As a result, it is possible to create a process group within another process group.

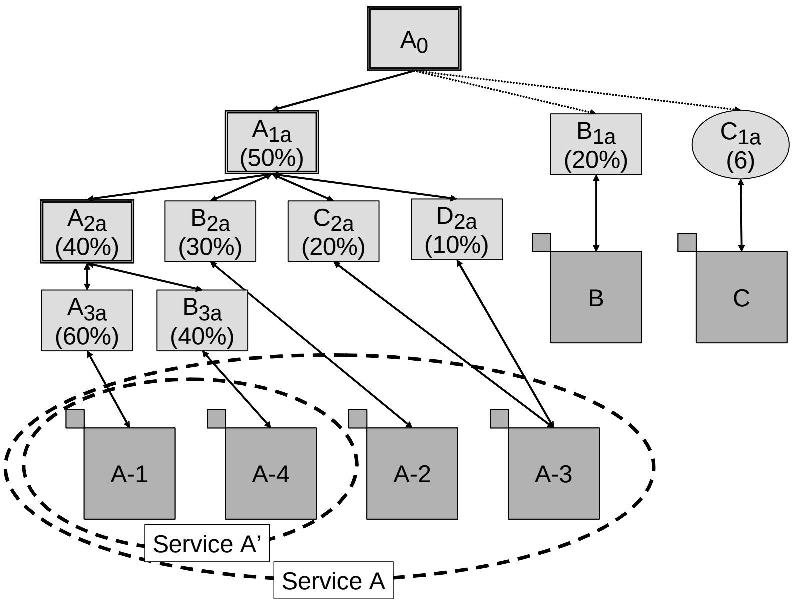

Figure 4. Relationships between a process group and executions

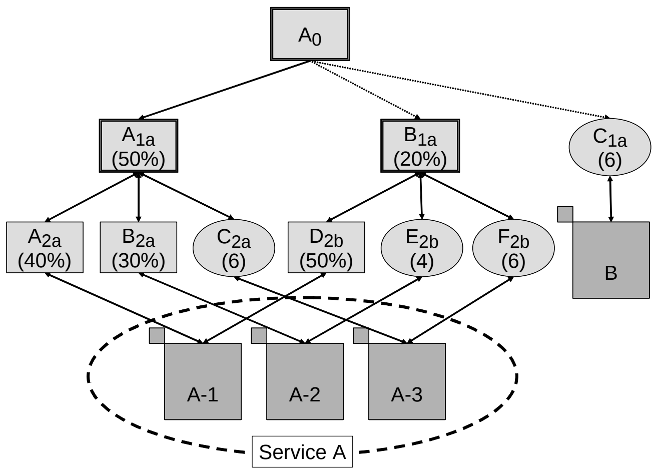

Figure 5. Two process group executions

Figure 5 shows a case where more than one execution is linked to a process group. When the second execution (B1a) is linked to a process group, leaf executions (D2b, E2b, F2b) have to be created and linked to each process (A, B, C) in the process group. As a result, each process within the process group is linked to two leaf executions.

3.4. Operation Interface of Execution Resource

We designed 8 operation interfaces for the execution resource for constructing an execution tree and controlling a program execution. Table 1 shows the interfaces

Table 1. Operation Interfaces of the execution resource

Form	Contents of operation
creat_execution(mips)	Create the execution specified by mips and return the execution identifier execid. When mips is between 1 and 100 it signifies the performance regulation execution degree (as a percentage with the performance of the processor itself taken to be 100 percent), when it is 0 or negative is signifies the priority of the execution degree (the absolute value is the process priority). In this case the larger value signifies the higher priority.
delete_execution(execid)	Delete the execution execid.
attach_execution(execid, pid)	Associate the execution execid and the process pid. Here linking to an execution execid that is already associated with a process will result in an error.
detach_execution(execid, pid)	Remove the association between execution execid and process pid.
wait_execution(pid, chan)	Forbid the assignment of processor [time] to process pid and its associated execution[s]; this puts the process in the WAIT state. When pid is 0, the executions associated with the currently running process[es] are forbidden from having processor [time] assigned to them; this puts the process[es] in the WAIT state. chan is a channel identifier.

wakeup_execution(pid, chan)	Make it possible to assign CPU time to the process pid and its associated executions; this puts the process in the READY state. When pid is 0, make it possible to assign CPU time to the executions that are associated with processes that have been forbidden the assignment of CPU time on channel chan; this puts the processes in the READY state. When chan is 0, it signifies all channels.
dispatch(pid)	Run process pid.
control_execution(execid, mips)	Change the execution degree of execid to mips. mips is interpreted as in creat_execution.

4. Execution Resource with Upper Bound

4.1. Mechanism of Rate Limiting Using Execution Resources

With regard to the operation of an existing OS, based on the time-sharing technique, a process can use the CPU time according to its priority and without an upper bound. Therefore, the allocation of CPU time to other services is affected when two or more programs that demand infinite CPU time run simultaneously. In this case, the service performance is significantly decreased.

To solve this problem, we propose an execution resource with an upper bound for the CPU usage ratio for preventing the abuse of the CPU resource. In this execution, the CPU time is allocated according to the priority until it reaches a specified ratio in a time slice. When it reaches the specified ratio, the state of a currently running process is changed to a WAIT state until the current time slice expires. Even if a process that is linked to an execution with an upper bound of the CPU usage ratio is affected by a malicious attacker, this system can prevent an attacker from using the CPU time. Moreover, the execution resource can be grouped with a user or a service. Therefore, the CPU usage ratio of a user or a service can be specified. As a result, the influence of a DoS attack can be controlled within the process group even if a child execution has been newly created because the execution belongs to the same group.

As described in a previous paper [7], [8], we can guarantee the performance of important processes by using execution resources with a good performance.

4.2. Advantages

(1) The proposed technique can prevent the use of a CPU resource by DoS attacks by using an execution with an upper bound of the CPU time.

(2) The CPU usage ratio that is employed by processes, users, and services can be controlled by using an access control mechanism for the execution resource.

(3) A maximum CPU usage ratio can be configured for each user and each service. This mechanism can be used for the prevention of CPU abuse in a manner similar to the quota mechanism in a file system.

5. Execution-Resource-Based Access Control Model for CPU Resource

5.1. Access Control of Execution Resource

The execution resource model requires the operations of process creation, execution creation, and the linking of a process and an execution for executing a program. The purpose of this access control is to enforce the policy of rate limiting. Thus, an OS must check the policy before executing operations related to the setting or changing of the degree of CPU usage. Therefore, in order to enforce an access policy for the CPU usage ratio, the OS checks the operation of execution creation and the change in the degree of CPU usage for each user and each program domain in the kernel. In this check, the OS calculates the degree of CPU usage before executing the operation. If the degree of CPU usage is permitted by the policy, OS grants the operation. Otherwise, OS refuses the operation.

5.2. Mapping Model of Execution Resource

General OSs can limit the usage of the disk by each user by using a "quota" mechanism. In a similar way, an OS should limit and control the usage ratio of the CPU for each user and each program domain. In addition, SELinux labels program domains and object types. A secure OS should limit and control the usage ratio of the CPU for each program domain. Thus, we propose a mapping model of execution resources. The one is for users on general OSs, and the other is for program domains on secure OSs.

Access Control Model for User: Discretionary Access Control is used to control access by restricting a subject's access to an object in most OSs. A subject is a user in most case. Thus, the access control of CPU resource for each user is suitable for general OSs. Figure 6 shows an execution tree for access control for users. In this execution tree, a child of root execution represents an execution that corresponds to a user. In order to control the degree of CPU usage, an administrator can control the mips of execution resource that is child of root execution. In this figure, an administrator controls execution A1, B1 and C1 that are correspond to user A, B and C.

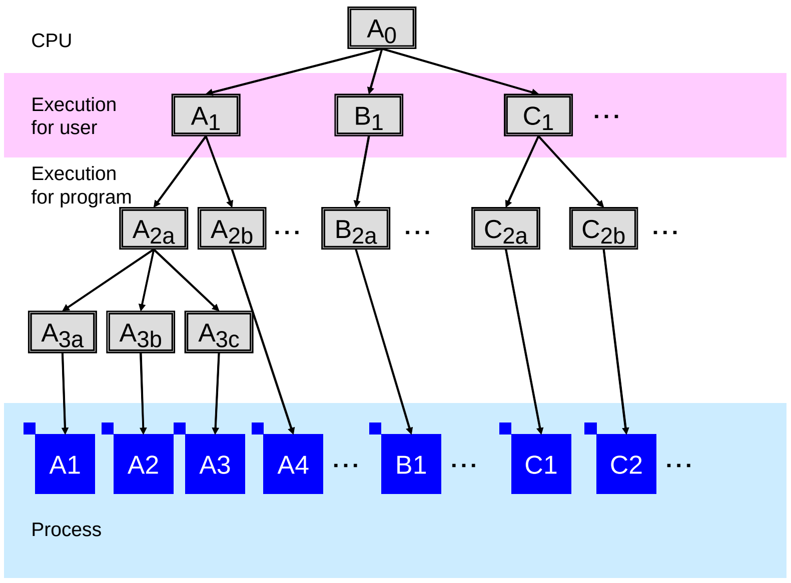

Figure 6. Execution Tree for the Access Control of Users

In order to construct the execution tree, an OS creates an execution resource as a child of root execution after login of a user. If the user executes a program, the OS create an execution as a child of the execution resource. Then, the OS links the execution to a process. To do this, the OS can create an execution tree for access control of users. This execution tree enables the OS to control the degree of CPU usage of each user.

An example of a policy is described as follows. A line of the policy file includes statement of allow, a user name, a maximum guaranteed performance, a maximum priority and an upper bound. Default policy should not allow any user to use any CPU resource.

```
# allow [username] {[maximum guaranteed performance] [maximum priority] [upper bound]}

allow userA {10 2 90}
allow userB {0 6 30}
```

Access Control Model for Program Domain: Subjects and objects in SELinux are labeled with security labels. The labels of subjects are called domains, while those of objects are termed types. Using these labels, SELinux can enforce a policy in a system. Because our purpose is to control the usage ratio of the CPU resource for each program domain, we propose an execution tree for the program domains. In order to control the degree of CPU usage for each program domain, we construct an execution tree as shown in Figure 7. In this model, a child execution resource of a root execution resource represents the execution resource of each program domain. In the figure, executions A1, B1, and C1 represent the execution resource of the domains http_t, sshd_t, and sysadm_t. Child execution resources of a program domain represent execution resources of the program (processes and process groups). These execution resources are linked to processes or process groups.

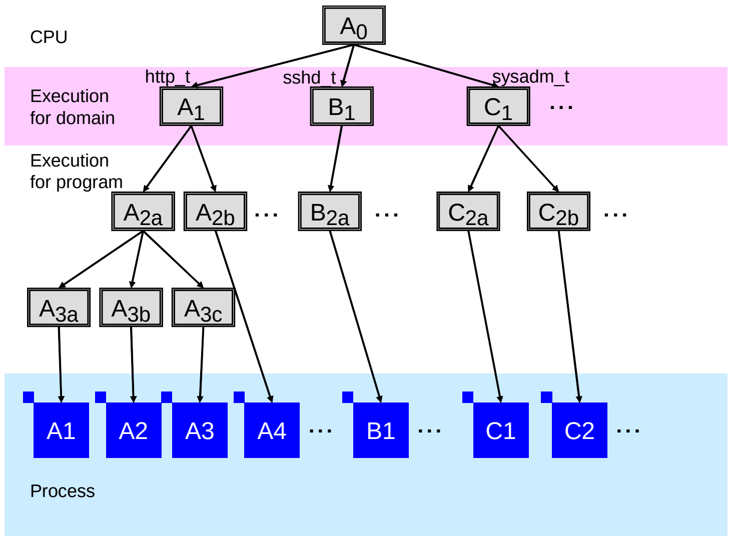

Figure 7. Execution tree for the access control of program domains

In the construction of an execution resource, the new execution resource must belong to same program domain as its parent process if domain transition does not occur. Thus, the new execution resource is linked to the execution resource that belongs to the program domain of the parent process. The administrator can control the usage ratio of the CPU resource for controlling access to these execution resources. In addition, a policy for enforcing CPU usage can be written in a policy file. An example of a policy that can be employed in secure OSs is as follows.

```
# allow domain {[maximum guaranteed performance] [maximum priority] [upper bound]}

allow http_t {0 1 80}
allow sysadm_t {10 0 100}
```

6. Implementation

We implemented the proposed mechanism on The ENduring operating system for Distributed EnviRonment (***Tender***) [14]. We implemented the execution resource as a resource of ***Tender***.

6.1. *Tender* Operating System

Tender, objects to be controlled and managed by the OS are called "resources." We realized a high modularity amongst the resources. The resources are provided with resource names and resource identifiers. Figure 8 shows the structure of a resource identifier and resource names. Additionally, the interface to the operation of resources is unified. Furthermore, the program sections that operate each resource are separated and shared programs are eliminated. The management information of each resource is also maintained separately. In addition, a pointer between the management tables of each resource is prohibited. As a result, each resource can exist irrespective of the existence of other resources.

We also implemented a BSD/OS-compatible system call interface in ***Tender***. Thus, ***Tender*** can execute binary programs that are identical to those of the BSD/OS.

6.2. Implementation of Execution Resource with Upper Bound

We added an upper bound function to the execution with priority in the Tender operating system. Figure 9 shows the flowchart of the processing of the scheduler that is related to the function. We added an entry that is a counter, indicating the usage of time slots for each execution with priority in the process management table. The scheduler of ***Tender*** increments the entry of a currently running process at the boundary of the time slots, and it then checks the number of the counter. If the number of the counter is equal to the usage-limited number of the time slot, the state of the current process is changed to a WAIT state. When the current time slice expires, the state of processes that have reached the upper bound is changed to a READY state.

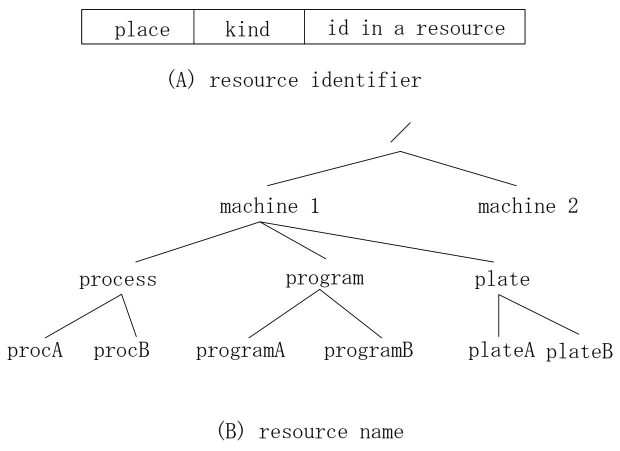

Figure 8. Resource identifier and resource name

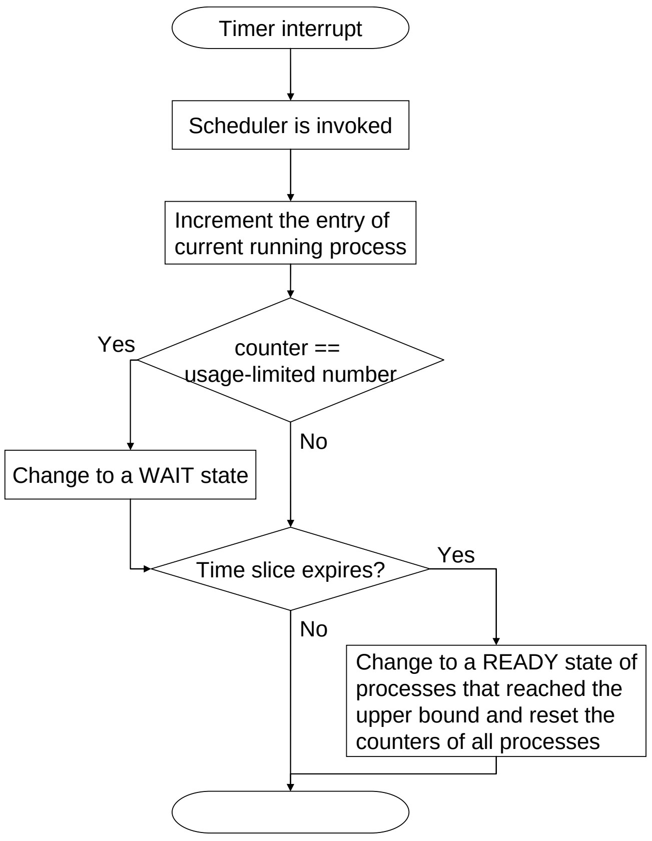
Timer interrupt
Scheduler is invoked
Increment the entry of current running process
counter == usage-limited number
Yes
No
Change to a WAIT state
Time slice expires?
Yes
No
Change to a READY state of processes that reached the upper bound and reset the counters of all processes

Figure 9. Flowchart of the processing of a counter of an upper bound

7. Evaluation

7.1. Purpose of experiments

We described the design, the implementation and the advantages of the execution resource with an upper bound. In this section, we describe the evaluation results of the execution resource.

The purpose of the evaluation is to make clear how the execution resource with an upper bound works by employing application programs. Furthermore, we should evaluate the basic experiment to understand the characteristics of the execution resource. For this reason, in order to evaluate the proposed mechanism, we performed two environments as follows.

(1) The experiment in local machine to evaluate basic performance.

(2) Simulation a DoS attack against Apache Web server.

In the local machine experiment, we used two simple programs. They are so simple that it is easy to analyze the results. In the simulation, we used Apache web server and a simple attack program. We evaluated the response time of the Apache web server to evaluate how the execution resource mitigates the DoS attack in a local machine.

7.2. Basic performance evaluation

We used two programs on a computer (CPU: Celeron 2.8GHz, OS: ***Tender***). The one is a normal process, and the other is an attacker's process. The normal program executes a procedure using CPU instructions only. The attacker's process executes an infinite loop to demand infinite CPU time.

Figure 10 shows processes and execution resources used in this experiment. The experiment is performed in two cases. In the first case, both the normal process and the attacker's process are linked to execution with priority which value is 4, and do not have upper limits. The priority means that a small number has high priority than a large number. 0 means the highest priority and 255 means the lowest priority. In the second case, the attacker's process is linked to execution with priority with upper limit. The value of the priority is 4, and the upper bound of execution is 30%. On the other hand, the normal process is linked to execution with priority, and its value is 6.

To perform the experiment, we executed the attacker's process first. Then we executed the normal process and measured the processing time of it. We also changed the number of attacker's processes from 0 to 7. Figure 11 shows the evaluation result of the experiment.

The processing time of the normal process increases as the number of attacker's processes increases. When the execution of it does not have an upper limit, the increase of processing time is large. But the execution of the attacker's process has the upper bound, the increase is suppressed. The processing time is less than that has no upper bound. This demonstrates that the tolerance to a DoS attack involving the usage of CPU time is CPU time is significantly improved in the proposed method.

7.3. DoS attack simulation against the Apache Web server

Next, we applied the proposed method to an application program. We choose the Apache web server as the target application program, because the Apache is used all over the world and the availability is required to web services.

The scenario of the simulation is as follows. We assumed that an attacker intrudes into a server machine (CPU: Celeron 2.8GHz, OS: ***Tender***) and takes control of the processes. Then, the attacker creates processes and attempts to use the CPU time in the machine. To use the CPU time, the attacker runs processes that execute an infinite loop. As a result, the performance of the web server (Apache) is influenced by the attack.

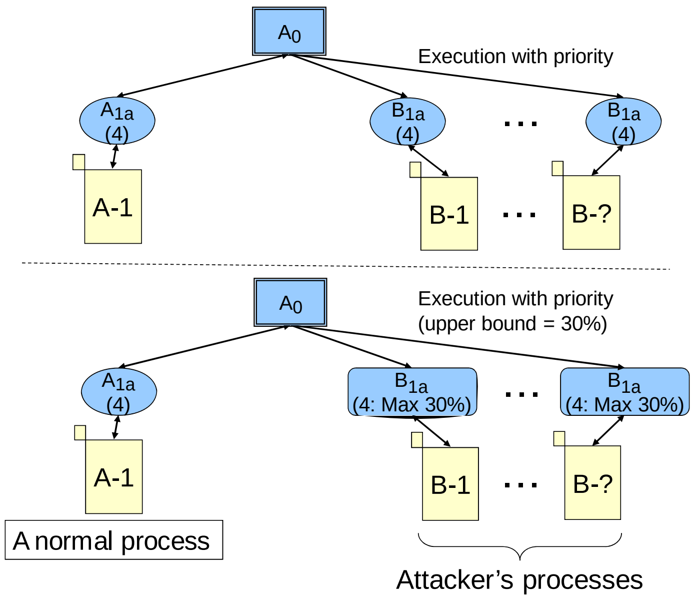

Figure 10. Processes and execution resources in the basic performance evaluation

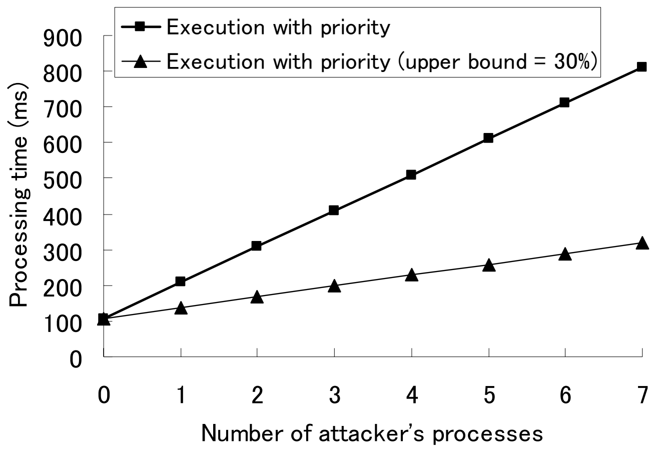

Figure 11. Experiment results in the basic performance

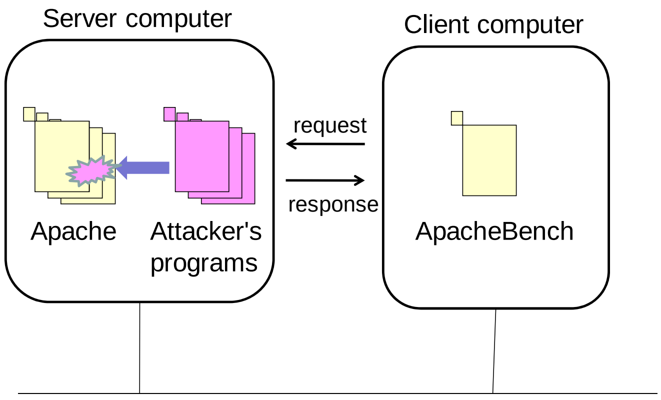

Figure 12. Experiment environment of a DoS attack simulation employing the Apache web server

Figure 12 shows the experiment environment of DoS attack simulation employing the Apache web server. To measure the server performance, we used ApacheBench as a benchmark program. ApacheBench is a command line program for measuring the performance of web servers. The ApacheBench comes bundled with the standard Apache source distribution. The ApacheBench was run on a client machine (CPU: Pentium4 3.0 GHz, OS: FreeBSD 4.3-RELEASE). We calculated the average response time of the Apache web server.

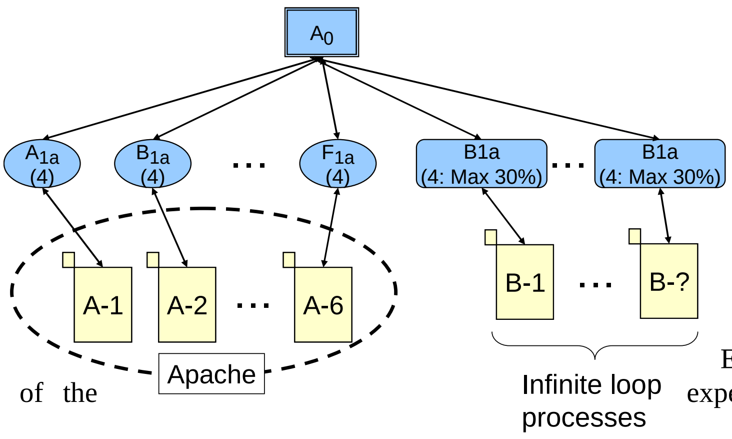

Figure 13. Example of the execution tree of the experiments

We performed two experiments. We tested the performance with different parameters. Figure 13 shows the example of the execution tree used in these experiments.

In the first experiment, the number of infinite loop processes was changed from 0 to 7. The priority of the web server was 4. The priority of the infinite loop processes was changed from 2 to 6. The upper bound of the execution resource linked to the infinite loop processes was 30%. We performed the same experiment, when the execution, which is linked to the processes, has no upper bound. We performed this experiment to evaluate the effectiveness of our proposed method against a DoS attack.

In the second experiment, the upper bound of execution was changed from 10% to 90%. The number of the infinite loop process is 1 and the priority of the web server was 4. The priority of the infinite loop processes was also changed from 2 to 6. The relation between the effectiveness of an upper bound and the performance of process is tradeoff. Thus, we must evaluate the characteristic of the tradeoff and show how to use the proposed method effectively.

Experiment 1: We performed this experiment for two cases. In the first case, the infinite loop processes were linked to an execution with priority. In the second case, the infinite loop processes were linked to an execution with an upper bound. The upper bound of the execution resource was 30% in this experiment. Figure 14 and Figure 15 show the results of this experiment.

When the priority of an infinite loop process equaled 2, the process had a priority that was higher than that of the Apache server. The scheduler of ***Tender*** does not recalculate the priority of the processes in which the CPU is used for a long time. Thus, it is not possible for any other process with a priority lower than that of an infinite process to run. Therefore, the Apache server was not able to respond to any process when execution resources with priority were linked in Figure 14.

The response time of the Apache server increases as the number of infinite loop processes increases in Figure 14, when the priority of the attacker's execution equaled 4. The response time for seven processes was greater than that in the absence of a process. Thus, the influence of the DoS attack was clearly apparent. The time of a time slice in ***Tender*** is 100ms. Thus, the response time increased 100ms when an infinite loop process was added. When the priority equaled 6, there was little influence to the Apache web server, because any infinite loop process was not able to run in this case.

On the other hand, even if the number of infinite loop processes increased for the proposed method in Figure 15 (priority = 2 or 4), the ratio increased only slightly. Although in the experiment of Figure 14, the Apache was not able to make response to any request, the Apache was able to make response by employing the execution resource had an upper bound. This result shows the effectiveness of the proposed method. When the priority equaled 6, there was little influence to the Apache web server.

When the number of processes was increased to seven, the ratio was increased by approximately 257%. The processing time was less than 10ms, thus it was enough to provide a service. In the proposed method, the tolerance to the DoS attack involving the use of CPU time has significantly improved by limiting the upper bound performance of the infinite loop process to 30%.

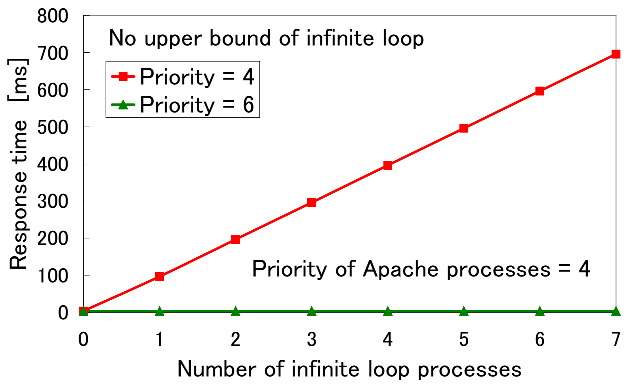

Figure 14. Result of the experiment 1 where the executions have no upper bound

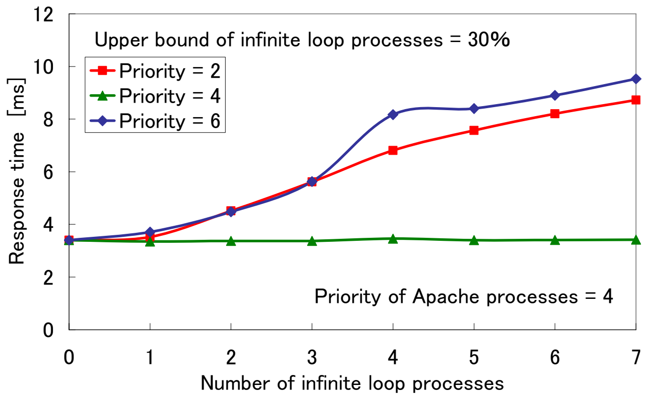

Figure 15. Result of the experiment 1 where the executions have an upper bound

Experiment 2: The response time was measured when one infinite loop process had a priority that was 2, 4, and 6. The number of infinite loop processes equaled 1. Figure 16 shows the response time when the priority of the infinite loop process is changed from 10% to 90%

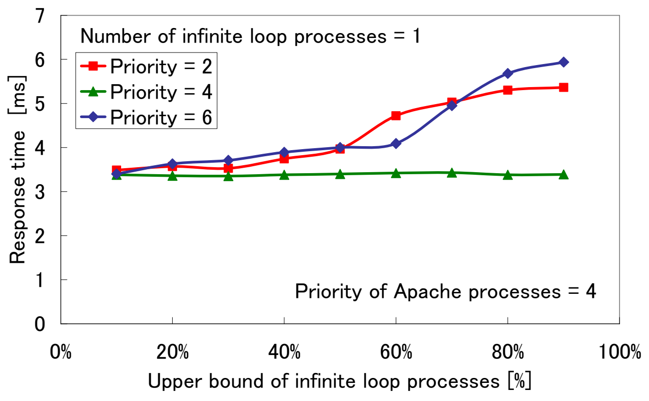

Figure 16. Result of the experiment 2

When the number of the infinite loop process equaled 1 and the execution of the process had no upper bound, the processing time increased apparently in Figure 14. In the case that the priority equaled 2, Apache was not able to make response to a client. In the case that the priority equaled 4, the response time was approximately 100ms. From the Figure 16, the response time was less than 7ms in all cases. These results show that the proposed method can suppress the influence of the infinite loop process. In addition, the server was able to respond to client programs by using the proposed technique. Therefore, the execution resource with an upper bound can limit the performance of the infinite loop process. Moreover, the response time increased slightly in proportion to the upper bound performance. When the priority equaled 6, there was little influence to response time.

From the fact described above, we may conclude that the increase in the Apache response time can be suppressed to less than 7ms, even if the infinite loop process has a high priority is executed. The proposed technique has more tolerance to an attack as compared to the original.

7.4. Evaluation of performance regulation

disadvantage of the proposed method is that it limits the performance of target services. In order to evaluate the influence, the performance of the Apache web server was measured by using execution with performance where the performance was regulated. Figure 17 shows an execution tree used for controlling the performance of the Apache web server. In this experiment, there was no process that had effects upon the performance of the Apache web server. We controlled the degree of CPU usage of execution resource A1a.

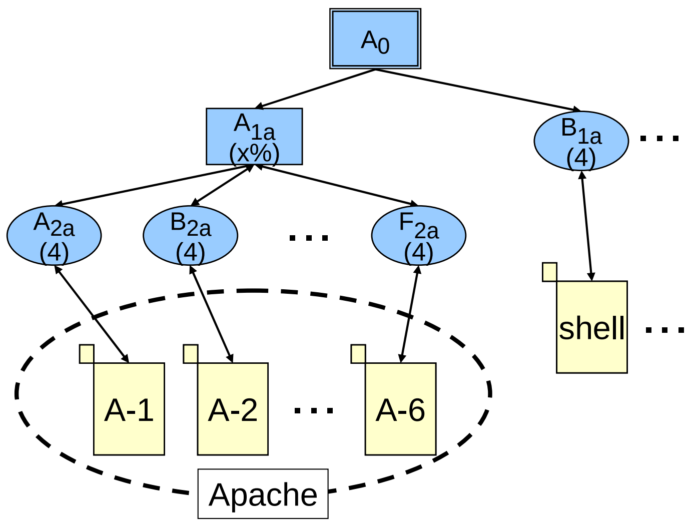

Figure 17. Execution tree used for controlling the performance of the Apache web server

Figure 18 shows the results of this experiment. When the performance was changed from 100% to 10%, the response time increased from 40%. This is because Apache did not require much CPU time for providing the service. Maybe the CPU usage ratio was less than 50% in this experiment. In the experiment 2 described above, the proposed method shows the good result against the DoS attack and the response time of Apache increased slightly compared with that of 100%. Therefore, to mitigate DoS attacks, it is not necessary to designate the low upper bound. If the upper bound is designated 90%, the response time is almost same as that of 100%. In addition, the proposed method can mitigate DoS attacks when the upper bound of an execution resource is 90%. Therefore, the Apache web server can get the both advantage of the performance and the tolerance

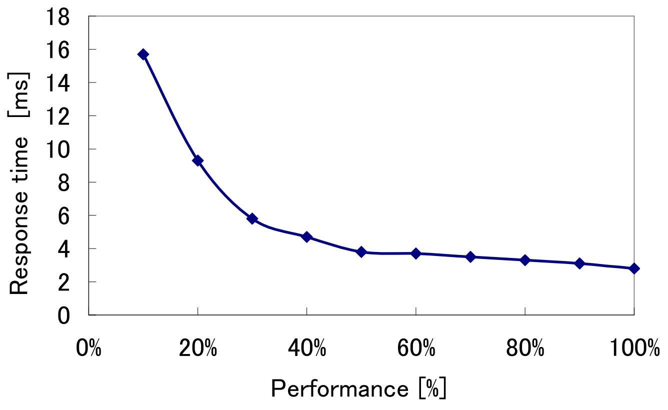

Figure 18. Response time of the Apache where the performance of the Apache was regulated by executions with performance

8. Conclusion

In this paper, we proposed an execution resource that controls the maximum extent of CPU usage for preventing the abuse of CPU resources. This execution resource enables system administrators to control the maximum extent of CPU usage for each process because the access control of CPU usage is performed in a similar way as an object such as a file.

In addition, we propose an access control model for CPU resources based on an execution resource for enforcing the access control policy. By mapping the execution resource to users and program domains, we can control the CPU usage of the users and the program domains. This model can be easily applied to most OSs because it requires only a modification of the process scheduler and process creation interface. The proposed mechanism is very useful for a computer with limited CPU power, such as mobile and ubiquitous environments.

Next, we describe the results of a basic performance experiment and a DoS simulation experiment employing the Apache web server. The results of the basic performance experiment show that the processing time of the normal process increases as the number of attacker's processes increases. When the execution of it does not have an upper limit, the increase of processing time is large. However, when the execution has the upper bound, the increase is suppressed.

The results of the DoS simulation experiment show that the ratio is increased by approximately 257%, when the number of processes is increased to seven. The processing time is less than 10ms, thus it is enough to provide a service. They also demonstrate that the tolerance to a DoS attack involving the usage of CPU time is significantly improved in the proposed method. This result shows that our proposal can ensure a good quality of web server. The second experiment shows that the increase in the Apache response time can be suppressed to less than 7ms, even if the infinite loop process with a high priority is executed. This result also reveals that the proposed technique has an increased tolerance to an attack as compared to the original execution.

Finally, in order to evaluate the influence, the performance of the Apache web server was measured by using executions with performance where the performance was regulated. The result shows that the response time is almost same as that of 100% where the upper bound is designated 90%, and the proposed method can mitigate DoS attacks. Therefore, the Apache web server can get the both advantage of the performance and the tolerance.

Acknowledgement: We would like to thank Koichi Sakai of Okayama University for his support. This research has been partially supported by the Kayamori Foundation of Informational Science Advancement and the research grant of Okawa foundation for information and telecommunications.

9. References

[1] “CERT/CC Statistics 1988-2005,” http://www.cert.org/stats/.

[2] D. Wagner and D. Dean, “Intrusion Detection via Static Analysis,” Proc. of the 2001 IEEE Symposium on Security and Privacy, pp. 156-168, 2001.

[3] S. A. Hofmeyr, S. Forrest and A. Somayaji, “Intrusion Detection using Sequences of System Calls,” Journal of Computer Security, Vol. 6, No. 3, pp. 151-180, 1998.

[4] R. Sekar, M. Bendre, P. Bollineni and D. Dhurjati, “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors,” Proc. of IEEE Symposium on Security and Privacy", pp. 144-155, 2001.

[5] “Security-Enhanced Linux,” http://www.nsa.gov/selinux/.

[6] R. S. Sandhu, E.J. Coyne, H.L. Feinstein and C.E. Youman, “Role-Based Access Control Models,” IEEE Computer, Vol. 29, No. 2, pp. 38-47, 1996.

[7] T. Tabata, Y. Nomura and H. Taniguchi, “Guarantee of Service Processing Time of Process Group for Multimedia Application,” Proc. of Pan-Yellow-Sea International Workshop on Information Technologies for Network Era (PYIWIT'02), pp. 104-111, 2002.

[8] T. Tabata and H. Taniguchi, “A Mechanism of Regulating Execution Performance for Process Group by Execution Resource on Tender Operating System,” IEICE Transactions on Information and Systems, Vol. J87-D-I, No. 11, pp. 961-974, 2004. (in Japanese)

[9] A. Garg and A. Reddy, “Mitigation of DoS attacks through QoS regulation,” Proceedings of IEEE International Workshop on Quality of Service (IWQoS), pp. 45- 53, 2002.

[10] O. Spatscheck and L. L. Petersen, “Defending Against Denial of Service Attacks in Scout,” Proc. of 3rd Symp. on Operating Systems Design and Implementation, 1999.

[11] A. Miyoshi and R. Rajkumar, “Protecting Resources with Resource Control Lists,” Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01), pp. 85-94, 2001.

[12] G. Banga, P. Druschel, and J. C. Mogul, “Resource containers: A new facility for resource management in server systems,” Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), pp. 45-58, 1999.

[13] J. Mirkovic and P. Reiher “A taxonomy of DDoS attack and DDoS defense mechanisms,” ACM SIGCOMM Comput. Commun. Rev., Vol. 34, No. 2, pp. 39-53, 2004.

[14] “Tender project,” http://www.swlab.it.okayama-u.ac.jp/lab/tani/research/tender-e.html

Authors

Toshihiro Tabata

Received the B.E. degree in 1998, the M.E. degree in 2000, and the Ph.D. degree in computer science in 2002, all from the Kyushu University, Fukuoka, Japan. In 2001 he was a Research Fellow of the Japan Society for the Promotion of Science. In 2002 he became a Research Associate in Faculty of Information Science and Electrical Engineering at Kyushu University. He has been an associate professor of Graduate School of Natural Science and Technology at Okayama University since 2005. His research interests include operating systems and computer security. He is a member of the Information Processing Society of Japan (IPSJ), the institute of Electronics, Information and Communication Engineers (IEICE), USENIX and ACM.

Satoshi Hakomori

Received the B.E. and the M.E. degrees from Nagoya University in 1986 and 1988. He then joined the Research and Development Headquarters at NTT Data Corporation. He is now working in the R&D Strategic Planning section. His Research interests include operating systems, distributed processing, and information security. He is a member of the Information Processing Society of Japan (IPSJ), IEEE and ACM.

Kazutoshi Yokoyama

Received the B.E. in 1988, the M.E. degrees in 1990 from Hiroshima University, and the Ph.D. degree in 2006 from Okayama University, Japan. He joined the Research and Development Headquarters at NTT Data Corporation in 1988. His Research interests include operating systems and distributed processing. He is a member of the Information Processing Society of Japan (IPSJ).

Hideo Taniguchi

Received the B.E. degree in 1978, the M.E. degree in 1980, and the Ph.D. degree in 1991, all from the Kyushu University, Fukuoka, Japan. In 1980, he joined NTT Electrical Communication Laboratories. In 1988, he moved Research and Development headquarters, NTT DATA Communications Systems Corporation. He had been an Associate Professor of Computer Science at Kyushu University since 1993. He has been a Professor of Faculty of Engineering at Okayama University since 2003. His research interests include operating system, real-time processing and distributed processing. He is the author of "Operating Systems" (Shoko Publishing Co. Ltd.) etc. He is a member of the Information Processing Society of Japan (IPSJ), the institute of Electronics, Information and Communication Engineers (IEICE), Japan Society for Software Science and Technology, and ACM.