A Secure System-wide Process Scheduler across Virtual Machines

Hidekazu Tadokoro
Dept. of Math. and Comput. Sci.
Tokyo Institute of Technology
tadokoro@csg.is.titech.ac.jp

Kenichi Kourai
Dept. of Creative Informatics
Kyushu Institute of Technology
kourai@ci.kyutech.ac.jp

Shigeru Chiba
Dept. of Math. and Comput. Sci.
Tokyo Institute of Technology
chiba@is.titech.ac.jp

***Abstract*—Server consolidation using virtual machines (VMs) makes it difficult to execute processes as the administrators intend. A process scheduler in each VM is not aware of the other VM and schedules only processes in one VM independently. To solve this problem, process scheduling across VMs is necessary. However, such system-wide scheduling is vulnerable to denial-of-service (DoS) attacks from a compromised VM against the other VMs. In this paper, we propose the *Monarch scheduler*, which is a secure system-wide process scheduler running in the virtual machine monitor (VMM). The Monarch scheduler monitors the execution of processes and changes the scheduling behavior in all VMs. To change process scheduling from the VMM, it manipulates run queues and process states consistently without modifying guest operating systems. Its hybrid scheduling mitigates DoS attacks by leveraging performance isolation among VMs. We confirmed that the Monarch scheduler could achieve useful scheduling and the overheads were small.**

***Keywords*-virtual machines; server consolidation; DoS attacks; process scheduling; performance isolation**

I. Introduction

Server consolidation is widely applied to improve the resource utilization of server machines. Particularly, the virtual machine (VM) technology is promising for consolidating legacy systems. Multiple physical servers can be easily migrated to multiple VMs using physical-to-virtual conversion (P2V) tools. The administrators can continue to use legacy systems including operating systems (OSes) as is in VMs. One drawback is that using VMs makes it difficult to execute processes as the administrators intend. Since VMs share physical processors, the execution of processes in one VM affects that in the other VMs. For example, consider that a process was configured to run only at idle time [1]. After server consolidation, it can run when the VM for it is idle, but the other VMs may not be idle at that time. As a result, the process may prevent the execution of more important processes in other VMs.

Under server consolidation using VMs, system-wide process scheduling is necessary to schedule processes across VMs. A global scheduler can monitor and control all the processes in all VMs properly. However, there are two issues for achieving system-wide process scheduling. First, the virtual machine monitor (VMM) underlying VMs is a possible place of implementing such a global scheduler, but its VM scheduler cannot perform process scheduling. The VMM cannot recognize processes in guest OSes. Second, system-wide scheduling is vulnerable to a new type of denial-of-service (DoS) attacks. If the attackers compromise one of the VMs, they may be able to prevent the execution of processes in other VMs only by running specific processes. For example, a process that is executed only at idle time cannot run forever.

In this paper, we propose a secure system-wide process scheduler called the *Monarch scheduler*. Since the Monarch scheduler runs in the VMM and the VMM is isolated from VMs, the attackers in VMs cannot compromise the Monarch scheduler itself. The Monarch scheduler monitors the execution of all processes in the whole system and changes the process scheduling in guest OSes. To change the scheduling behavior of guest OSes from the VMM, the Monarch scheduler directly manipulates run queues or process states without modifying guest OSes. In addition, the Monarch scheduler performs hybrid scheduling to mitigate DoS attacks introduced by system-wide scheduling. The hybrid scheduling can take trade-off between system-wide scheduling and performance isolation among VMs by switching the controlled and autonomous modes frequently.

We have implemented the Monarch scheduler in Xen [2]. To access kernel data in guest OSes, the Monarch scheduler uses type information obtained from OS kernels and translates virtual addresses used by them into machine addresses used by the VMM. At this time, it guarantees that it can consistently access the kernel data by checking locks for mutual exclusion in guest OSes. To obtain the accurate execution time of processes, the Monarch scheduler monitors the switches of virtual address spaces corresponding to processes. From our experimental results, it was shown that the overheads due to the Monarch scheduler were small enough. Also, the Monarch scheduler could achieve useful scheduling policies, mitigating DoS attacks.

The rest of this paper is organized as follows. Section II describes the necessity and issues of system-wide process scheduling. Section III presents the Monarch scheduler. Section IV describes the implementation details. Section V shows experimental results using the Monarch scheduler. Section VI explains related work and Section VII concludes the paper.

II. System-wide Process Scheduling for VMs

Under server consolidation using VMs, it is often difficult to schedule processes as the administrators intend. All CPU resources were used by one OS before server consolidation while they become shared among VMs after that. For example, antivirus software is often executed at a lower priority [3], but higher-priority processes may be running in other VMs even when such processes are not running in the VM that is scanning viruses. As another example, file indexing by search engines is often executed at idle time [4], [5], but processes may be running in other VMs even when the VM that attempts to index files is idle. A process scheduler in each guest OS is not aware of processes in the other VMs because it behaves as it manages all processes in the whole system. As a result, such less important processes in one VM prevent the execution of more important processes in other VMs such as web servers and databases.

To solve this problem, system-wide process scheduling across VMs is necessary. If a global scheduler can know the existence of more important processes among all VMs, it can suppress the execution of less important ones. For example, antivirus software is executed so that higher-priority processes in all VMs are given more CPU time. File indexing is executed only when all the VMs are idle. To enable such scheduling, a global scheduler has to monitor and control the execution of all processes in all VMs. In a VM environment, the VMM underlying VMs is a possible place of implementing such a global scheduler. The VMM manages the whole system and schedules all VMs.

However, the VM scheduler of the VMM is not suitable for system-wide process scheduling. The VMM cannot recognize the process because the process is an abstraction of OSes. The VM scheduler can give priorities or reserve the CPU resource only to VMs. For example, it can give a lower priority to the VM that executes antivirus software. Virus scans are thereby executed at a lower priority in the whole system but higher-priority processes in the VM would not be given sufficient priorities as well. To allow the VMM to consider processes in guest OSes, new VM schedulers have been proposed [6], [7], but they require the modification of guest OSes to pass information on the priorities of processes to the VMM. When legacy OSes are used in consolidated servers, we cannot modify the OSes.

In addition, system-wide process scheduling has a security issue to be considered. It is inherently vulnerable to a new type of DoS attacks. Let us consider that the attackers compromise one VM and intrude it. The attackers may be able to perform DoS attacks against processes in other VMs only by running specific processes. For example, they can prevent the execution of file indexing by running one process with a busy loop. A global scheduler stops file indexing due to the process running in the compromised VM. Although the administrators should elaborate scheduling policies that can tolerate this type of DoS attacks, such policies become complicated and error-prone. Traditionally, performance isolation provided by VMs could prevent such attacks. If there are processes to be ready to run in a VM, the VM can receive a certain CPU time. System-wide process scheduling breaks such performance isolation.

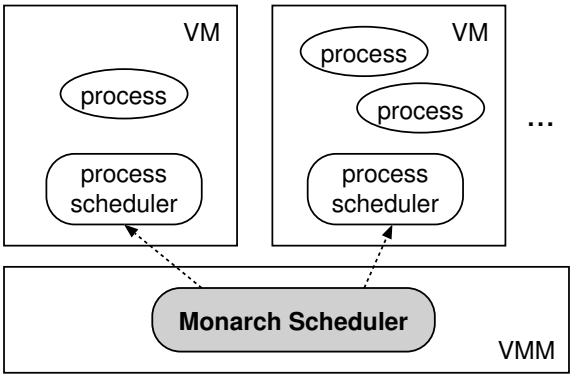


Figure 1. The Monarch scheduler running in the VMM.

Our threat model is as follows. We assume that the attackers intrude some of the VMs and perform DoS attacks specific to system-wide scheduling. We do not assume that the kernels of guest OSes and the VMM have been compromised because the VMM and security hardware can guarantee their integrity. [8]–[11]

III. Monarch Scheduler

A. Process Scheduling by the VMM

The Monarch scheduler is a secure system-wide process scheduler running in the VMM, as illustrated in Figure 1. Since the VMM is isolated from VMs on top of it, the attackers in VMs cannot compromise the Monarch scheduler. The Monarch scheduler monitors the execution of processes and changes the scheduling behavior for all VMs from the VMM. Normally, the VMM cannot obtain information in guest OSes or change the behavior because it is not aware of the internals of guest OSes. To understand guest OSes, the Monarch scheduler analyzes the memory of VMs using information on kernel data. Modifying guest OSes is not necessary so as to cooperate with the Monarch scheduler. The Monarch scheduler exploits the existing process schedulers in guest OSes as much as possible. Each process is basically scheduled by a process scheduler in the VM that it belongs to. The role of the Monarch scheduler is to watch the whole system consisting of multiple VMs and change the scheduling behavior in guest OSes so that a system-wide scheduling policy is achieved.

The Monarch scheduler changes the process scheduling in guest OSes by suspending and resuming processes. For this adjustment, the Monarch scheduler directly manipulates

run queues of process schedulers or process states in guest OSes. To suspend a process that is ready to run, the Monarch scheduler removes it from a run queue. Since a guest OS schedules only processes in run queues, any CPU time is not allocated to the removed process. At this time, the Monarch scheduler guarantees that a guest OS is not manipulating its run queues to keep the consistency. For a process that is blocked or currently running, the Monarch scheduler rewrites its state so that a guest OS does not schedule it later. To resume a suspended process, the Monarch scheduler rewrites the process state if necessary and inserts it into a run queue.

The Monarch scheduler records the execution time of processes in the VMM. The VMM cannot understand processes in guest OSes directly, but it can recognize them via their virtual address spaces [12]. The Monarch scheduler monitors the switches of virtual address spaces and measures the CPU time used by processes. Although the process time is also recorded by guest OSes, it may be inaccurate when processes are running in VMs. As far as an OS kernel is not modified to be aware of the VM, it cannot recognize the context switches between VMs. As a result, it may incorrectly account the CPU time to a process even while the VM for it is not scheduled. The measurement by the VMM does not depend on the accounting mechanisms in guest OSes.

B. Hybrid Scheduling

To mitigate inherent DoS attacks introduced by system-wide process scheduling, the Monarch scheduler provides hybrid scheduling. The hybrid scheduling periodically switches two modes: controlled and autonomous. In the *controlled* mode, the Monarch scheduler performs system-wide process scheduling. In the *autonomous* mode, it stops its own scheduling and allows the VMM and guest OSes to perform their own original scheduling. Even if the attackers in compromised VMs run malicious processes so that victim processes in other VMs are suspended by the Monarch scheduler, such DoS attacks are mitigated thanks to the autonomous mode. The victim processes can run for a certain period at least.

The hybrid scheduling can achieve both system-wide scheduling and performance isolation among VMs. The controlled mode allows process execution that is not aware of the isolation enforced by VMs. The autonomous mode mitigates DoS attacks across VMs by the enforcement of performance isolation. To take trade-off between two modes, the Monarch scheduler allows the administrators to adjust the ratio of scheduling between these two modes. As scheduling time in the controlled mode becomes longer, the Monarch scheduler can control process execution more accurately. As that in the autonomous mode becomes longer, performance isolation among VMs is left more largely.

IV. Implementation

We have implemented the Monarch scheduler in Xen 3.4.2 [2]. In Xen, a regular VM is called *domain U*. In the current implementation, the Monarch scheduler supports the Linux 2.6 guest OS for the x86-64 architecture.

A. Scheduler Overview

The Monarch scheduler is invoked by timer interrupts in the VMM. The default interval is 10 ms in the current implementation. First, the Monarch scheduler pauses all virtual CPUs to stop the execution of all domain Us. This prevents the conflicts of process scheduling between the Monarch scheduler and guest OSes. During the controlled mode in hybrid scheduling, the Monarch scheduler traverses the process lists in guest OSes and finds target processes. Based on the execution time of the processes, it changes process scheduling in guest OSes by suspending or resuming some of the processes. Finally, it continues all domain Us again. During the autonomous mode, the Monarch scheduler does nothing.

When the Monarch scheduler switches between the controlled and autonomous modes, it largely changes process scheduling in guest OSes. When it enters the autonomous mode, it resumes all the processes that have been suspended by it. All processes are scheduled by process schedulers in guest OSes as if the Monarch scheduler is not used. When the Monarch scheduler comes back to the controlled mode, it re-schedules all processes again based on its scheduling policy and suspends processes if necessary. The periods allocated to these two modes can be configured by the administrators. The default periods are 500 ms for the controlled and autonomous modes, respectively.

B. Accessing Kernel Data

The Monarch scheduler obtains information on data types from the debug information of OS kernels. An example of such type information is the task_struct structure for representing the process in Linux. The debug information is generated by compiling the kernel with the debug option and stored in the DWARF [13] format. Such type information can be obtained from the source code of the kernel, but it is more complicated. The Linux kernel contains various macros for configuration and enables specific fields in data structures by defining macros.

To access data in a guest OS from the VMM, the Monarch scheduler have to translate their virtual addresses in domain U into *machine addresses*. In Xen, the VMM uses the machine address to access the entire memory. Domain U is given *pseudo-physical memory* for the illusion of its own physical memory. First, the Monarch scheduler looks up the page table in domain U and translates a virtual address into a pseudo-physical address in the domain U. Next, it looks up the *P2M table* in the VMM and translates the pseudo-physical address to a machine address. The Monarch

scheduler maintains the result of this translation as a cache. When it accesses virtual addresses in the same page, it can obtain machine addresses from the cache directly. The cache is invalidated before the Monarch scheduler continues the domain U because the page table and the P2M table can be changed while the domain U is running.

When the Monarch scheduler examines processes in a guest OS, it traverses a circular list including all the processes. The staring point of the process list is the init_task symbol, which is invariant in each kernel image. The virtual address of this symbol is also obtained from the symbol table in the debug information of the kernel. On the other hand, it is not so straightforward to find all the run queues in a guest OS. In Linux, a run queue is created for each virtual CPU. Since the number of virtual CPUs is not determined until a VM is created, the address of each run queue changes according to the number. Therefore, the Monarch scheduler obtains the address of a run queue by starting from the GS base register of a virtual CPU. The register points to per-CPU specific data structure named x8664_pda. This data structure contains a pointer to a run queue.

The Monarch scheduler guarantees to consistently access kernel data in guest OSes. If it inspects a run queue while a guest OS is modifying it at the same time, it and/or the guest OS may crash. To prevent this situation, the Monarch scheduler checks locks for kernel data. Linux uses spinlocks for mutual exclusion of accessing run queues and the process list, respectively. Before the Monarch scheduler access such kernel data, it checks whether the corresponding spinlock is acquired by a guest OS or not. If the spinlock is not acquired, the Monarch scheduler can safely manipulate kernel data. It does not need acquire the spinlock because it pauses the domain U in which the guest OS runs. If the spinlock is already acquired by a guest OS, the Monarch scheduler skips scheduling at that time. Since a guest OS should release such spinlocks in a short period, the Monarch scheduler can perform scheduling shortly.

C. Suspending/Resuming Processes

The Monarch scheduler uses several techniques for suspending processes according to their state. In Linux, a process has three main states: ready, running, and blocked. For a process in the ready state, the Monarch scheduler manipulates a run queue to suspend it. A process in this state is waiting for being scheduled in a run queue. In Linux, a run queue is an array of lists, each of which accommodates runnable processes with the same priority, as shown in Figure 2. To suspend a process in this state, the Monarch scheduler removes the process from one of the lists in a run queue. Together with this manipulation, it also updates the counter that maintains the number of processes in a run queue. Since the processes in run queues do not hold any locks in the kernels, suspending them does not cause deadlocks.

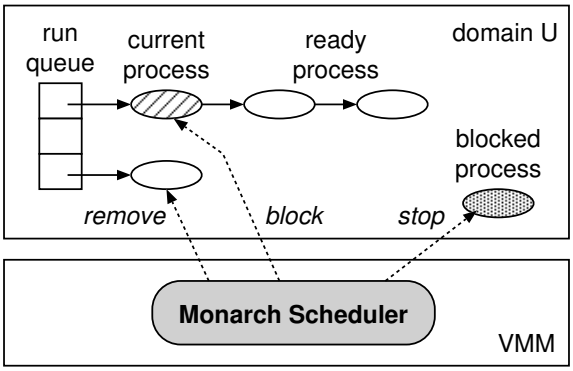


Figure 2. Suspending processes in various states.

For a process in the running or blocked state, the Monarch scheduler rewrites the process state to suspend it. A process in the running state is the currently running process and a process in the blocked state is waiting for I/O completion or lock acquisition. To suspend a process in the running state, the Monarch scheduler changes the process state to blocked. When a process scheduler in a guest OS is invoked for a context switch, the current process is removed from a run queue. If the state is not running, the process is not inserted into the run queue again. At this time, the process does not hold any locks in the kernel. Although the current process is also in a run queue in Linux, the Monarch scheduler cannot suspend it by directly removing it from the run queue. Even if the Monarch scheduler does so, the process scheduler in a guest OS inserts the current process into the end of the run queue and schedules it again.

On the other hand, the Monarch scheduler changes the process state to stopped for a process in the blocked state. The state of stopped is identical to that of a process suspended by the SIGSTOP signal. Even when a process is woken up by an event such as I/O completion, the wake-up function does not insert the process into a run queue if the process state is stopped. When such a process is stopped by a guest OS, it is guaranteed that it does not hold any locks in the kernel. Although the process can be resumed by sending the SIGCONT signal, the Monarch scheduler suspends the process again immediately. Since a process in this state is not in a run queue, the Monarch scheduler cannot remove the process from a run queue. Conversely, this process rewriting is ineffective for a process in the ready state. Even if the Monarch scheduler rewrites the process state, a process scheduler in a guest OS schedules the process without checking its state.

To resume a process that has been removed from a run queue, the Monarch scheduler inserts it into the run queue from which it has been removed. The inserted list in the run queue is selected according to its priority. For a process whose state has been rewritten, the Monarch scheduler rewrites its state to the ready state before inserting it into a

run queue.

D. Monitoring Accurate Process Time

To record the execution time of each process, the Monarch scheduler measures the CPU time used for the execution in the context of the corresponding virtual address space. A virtual address space is uniquely identified by the machine address of the page directory in a page table. When a guest OS sets the address to the CR3 register in a virtual CPU for the context switch between processes, the Monarch scheduler can check the address. The instruction for changing CR3 is privileged and trapped by the VMM. Similarly, the Monarch scheduler can check the address when a context switch occurs between VMs and the current virtual address space is changed. The CPU time from when the specific value is set to CR3 until the value of CR3 is changed is accumulated as the corresponding process time.

The Monarch scheduler binds virtual address spaces to real processes by using process information in guest OSes. In Linux, the address of the page directory is stored in the mm_struct structure, which is followed from task_struct. By traversing the process lists in guest OSes, the Monarch scheduler can bind the accurate execution time to each process.

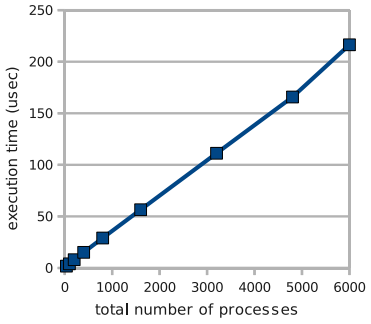
V. Experiments

We performed experiments to examine the overheads and the scheduling abilities of the Monarch scheduler. For a server machine, we used a PC with one Intel Core 2 Duo processor E6600, 6 GB of memory, and a Gigabit Ethernet NIC. We ran Xen 3.4.2 for the x86-64 architecture on this PC. For domain 0, we allocated two virtual CPUs and 1 GB of memory and we ran Linux 2.6.18. For domain U, we allocated one virtual CPU and 1 GB of memory and we ran Linux 2.6.18 as a guest OS. For a client machine, we used a PC with one Intel Core 2 Quad processor Q9550S, 8 GB of memory, and a Gigabit Ethernet NIC. These two machines were connected with a Gigabit Ethernet switch.

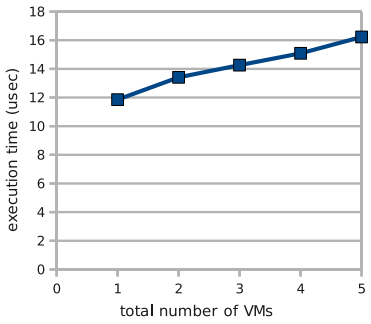
A. Scheduling Overheads

To examine the overheads of running the Monarch scheduler, we measured the time needed for traversing the process lists in guest OSes. Each VM is paused during this traversal. In this experiment, the Monarch scheduler searched target processes from the process lists by comparing process names and did not suspend or resume any processes. On each guest OS, 36 processes were running originally. We performed the traversal of the process lists 1000 times and obtained the average time.

First, we changed the number of processes in one VM between 36 and 6000 to examine the impact of the length of the process list. Spawning more than 6000 processes caused an out-of-memory error. To adjust the number of processes, we ran dummy processes that always slept. As in Figure 3(a),



(a) For processes



(b) For VMs

Figure 3. The time for traversing process lists.

the traversal time is proportional to the number of processes and 36 ns for one process. For traversing 6000 processes, it takes 220 μs and the overhead is 2.2% when the scheduling interval is 10 ms. However, running 6000 processes in one VM is not realistic. For 400 processes, it takes 15 μs and the overhead is 0.15%.

Next, we changed the number of VMs between one and five. The purpose of this experiment is to clarify the overheads of inspecting multiple VMs. Therefore, we fixed the total number of processes in the whole system to 300. Figure 3(b) shows that the time for traversing 300 processes depends on the number of VMs but increases only 0.88 μs per VM. This overhead comes from increasing the number of pausing virtual CPUs and checking locks for kernel data. From this result, the scheduling overheads mainly come from the number of processes.

B. Monitoring Overheads

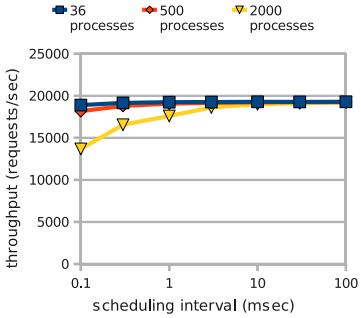
To examine the overheads of monitoring process execution in the VMM, we measured the time needed for recording the execution time of processes with CR3 and the number of context switches per second. We performed this experiment at the VM start-up time and in a steady state. We regarded 15 seconds after booting a VM as the VM start-up time. While many processes were created at the VM start-up time, only a few processes ran in a steady state. We used between one and five VMs.

At the VM start-up time, it took 0.26 μs per context switch and context switches occurred 1467 times per second on average. From these results, the overhead of process monitoring is 0.04%. In a steady state, on the other hand, it took 0.20 μs per context switch. Since context switches occurred 129 times per second, the overhead of process monitoring is 0.003%. The reason why it takes more time at the start-up time is that newly created processes need to allocate new recording area. In any cases, this overhead is negligible.

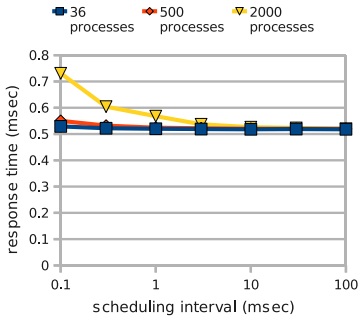
C. Performance Degradation

To examine the performance degradation due to the above scheduling and monitoring overheads, we measured the throughput and response time of the lighttpd web server [14]. In this experiment, we created one VM and ran the lighttpd process and dummy processes. The Monarch scheduler traversed the process list to find target processes and did not change the process scheduling. We used the ApacheBench benchmarking tool [15] and sent ten requests concurrently.

We changed the scheduling interval at which the Monarch scheduler was invoked between 0.1 and 100 ms. We measured the throughput and response time when the number of processes was 36, 500, and 2000. We chose the maximum number of processes so that the time for traversing the process list was less than 0.1 ms. Figure 4 shows the results. The performance was degraded largely when the interval was 0.1 ms and the number of processes was 2000. However, this interval is too short realistically. When the interval was 10 ms, which is the default in the Monarch scheduler, the throughput was degraded by 1.5% and the response time became 1.3% longer even for 2000 processes. For 500 processes, the performance was degraded by less than 0.3%.



(a) Throughput



(b) Response time

Figure 4. The performance degradation of a web server.

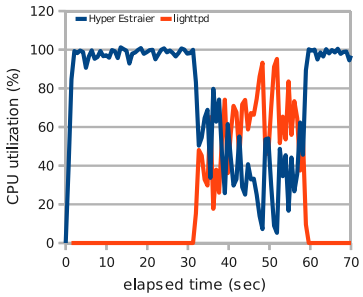
D. System-wide Idle-time Scheduling

We examined the effectiveness of idle-time scheduling achieved by the Monarch scheduler. We ran lighttpd in VM 1 and the file indexing in Hyper Estraier [16] in VM 2. Hyper Estraier is a high-performance text search engine. First, we monitored the activities of these two processes when we did not use the Monarch scheduler. Figure 5(a) shows the changes of the CPU utilization of these two processes. While lighttpd was running in VM 1, the file indexing was also running because it was only an active process in VM 2. Therefore, the file indexing largely affected the execution of lighttpd although it should stop. The throughput of lighttpd was degraded by 24% and the response time was 32% longer.

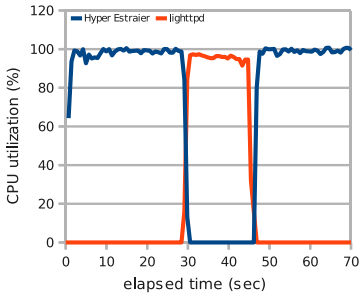
Second, we used the Monarch scheduler to execute the file indexing only at idle time in the whole system. To examine the accuracy of scheduling, we disabled hybrid scheduling in this experiment. Figure 5(b) shows the results of this system-wide process scheduling. When lighttpd started running in VM 1, the file indexing stopped immediately in VM 2. The throughput of lighttpd was degraded by 2.4% and the response time was 2.5% longer.

Third, we enabled hybrid scheduling of the Monarch scheduler. Without hybrid scheduling, the attackers can completely prevent the execution of the file indexing by making lighttpd always busy. We changed the ratio of the controlled and autonomous modes. Figure 6(a) shows the CPU utilization of the file indexing for the various ratios of the autonomous mode. As the ratio becomes large, the file indexing runs more. Figure 6(b) shows the changes of the CPU utilization of two processes when the ratio is 50%. Strictly speaking, hybrid scheduling violates idle-time scheduling but prevents DoS attacks from the VM that executes lighttpd. When the ratio of the autonomous mode is more than 80%, the CPU utilization increases steeply. This is because the Monarch scheduler switches from the controlled mode to the autonomous one before the manipulation of guest OSes works effectively.

Hybrid scheduling degrades the performance of lighttpd even when lighttpd runs normally. The performance degradation is shown in Figure 7. As the ratio of the autonomous mode is increasing, the throughput is decreasing and the response time becomes longer. When the ratio is 50%, the throughput degradation is 9.7% and the response time is 8.8% longer.

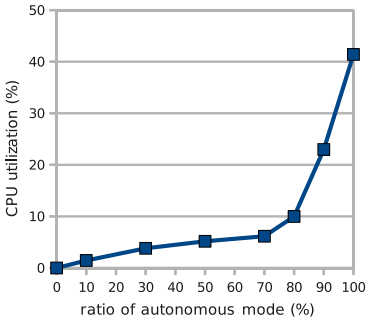


(a) Default scheduling

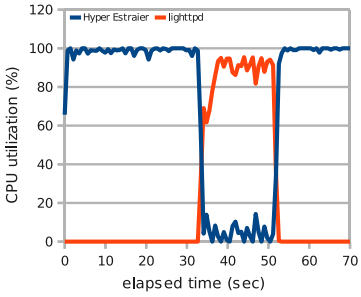


(b) Idle-time scheduling

Figure 5. System-wide idle-time scheduling for Hyper Estraier.



(a) CPU utilization



(b) Hybrid scheduling

Figure 6. The effects of hybrid scheduling with idle-time scheduling.

E. System-wide Priority Scheduling

We examined the effectiveness of priority scheduling by the Monarch scheduler. We ran the power test of the DBT-3 benchmark [17] in VM 1 and the virus scanner of ClamAV [18] in VM 2. DBT-3 tested the performance of PostgreSQL in a decision support system. Without system-wide priority scheduling, the virus scanner interfered with PostgreSQL across VMs as shown in Figure 8(a). When we ran only DBT-3, the power test took 221 seconds. On the other hand, it took 384 seconds when we ran the power test with the virus scanner.

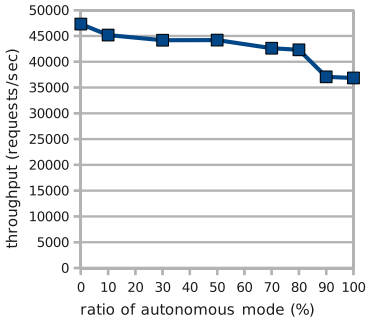
To execute the virus scanner in a lower priority than PostgreSQL, we configured the priorities of PostgreSQL and the virus scanner so that their CPU shares were 2 and 1, respectively. Figure 8(b) shows the results. When PostgreSQL needs much CPU time, for example, between 100 and 150 seconds and between 320 and 380 seconds, the CPU utilization of PostgreSQL is approximately double that of the virus scanner. When PostgreSQL is idle, the virus scanner uses most of the CPU time. Under this scheduling, the power test took 275 seconds, which was 28% faster than when we did not use the Monarch scheduler.

Next, we performed other experiments to demonstrate DoS attacks to ClamAV and show the effectiveness of hybrid scheduling. We ran four processes for MEncoder [19] in VM 1 and the virus scanner of ClamAV [18] in VM 2. MEncoder encoded MPEG-4 video data. We assigned a high priority to MEncoder and a low priority to the virus scanner so that their shares were 2 and 1, respectively. Figure 9(a) shows the total CPU utilization of all MEncoder processes and that of the virus scanner when we did not use the Monarch scheduler. The virus scanner could use more than 50% of the CPU time. However, when we used the Monarch scheduler, the CPU utilization of the virus scanner was reduced to 12%, as in Figure 9(b). As such, the attackers can perform DoS attacks using system-wide scheduling.

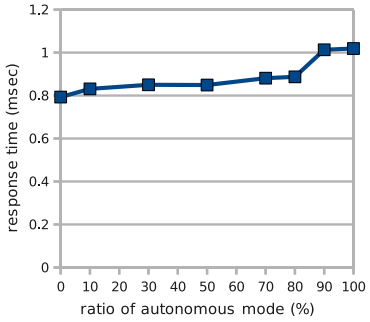
Figure 10(a) shows the CPU utilization of these two when we enabled hybrid scheduling. The virus scanner can obtain more CPU time. Figure 10(b) shows the relationship between the number of MEncoder processes and the CPU utilization of the virus scanner. Without hybrid scheduling, as the attackers ran more MEncoder processes in VM 1, the CPU utilization of the virus scanner in VM 2 was decreasing. When we configured the ratio of the autonomous mode to 50% in hybrid scheduling, the virus scanner can obtain more than 16% of CPU time even for more than four MEncoder processes.

F. Dependence on Guest OSes

The Monarch scheduler depends on the internal structures of guest OSes because it directly monitors and manipulates the kernel data. Even for the same OS, its internal structures can change among different versions. Several data structures are altered by refactoring, adding new features, changing the

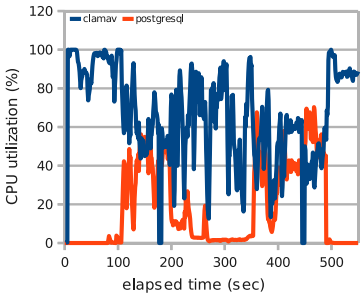


(a) Throughput

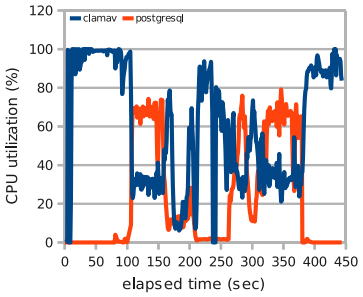


(b) Response time

Figure 7. The performance degradation by hybrid scheduling.



(a) Default scheduling



(b) Priority scheduling

Figure 8. System-wide priority scheduling for DBT-3 and ClamAV.

scheduling algorithm. To examine how much the Monarch scheduler has to be modified when the Linux kernel is updated, we inspected 33 versions of the Linux kernel 2.6.

The kernel was largely modified between these versions, but it was shown that the Monarch scheduler was not affected by most of kernel updates because it depends only on the scheduling and management of processes. When new data structures are added to an OS, the Monarch scheduler can ignore them if it does not refer to them. Even if data structures that the Monarch scheduler refers to are changed, the Monarch scheduler just obtains type information on them from the debug information of the kernel again.

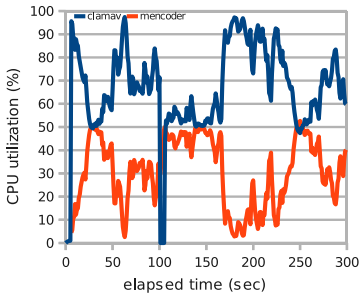
There were several small changes that we had to modify manually. In 2.6.14, a field of the spinlock_t structure was changed so that it was contained in another structure. In 2.6.18, the runqueue structure was simply renamed to rq. In 2.6.30, the calculation of the address of a run queue was changed so that a fixed offset was added to the value of the GS base register. For these changes, we could easily modify the Monarch scheduler.

On the other hand, the scheduling algorithm was changed from the O(1) scheduler to the completely fair scheduler (CFS) in 2.6.23. The O(1) scheduler uses doubly-linked lists of processes as run queues while CFS uses a red-black tree. Since the new scheduler changed both data structures and a scheduling algorithm, we needed to modify the Monarch scheduler largely. According to our deep inspection, we could modify the Monarch scheduler so that it can change the behavior of CFS.

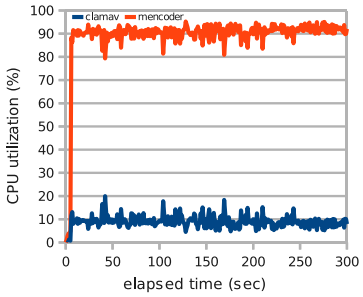
VI. Related Work

Researchers have proposed VM scheduling mechanisms that can give global priorities to processes across VMs. In guest-aware VM scheduling [6], each guest OS notifies the VMM of the highest priority among processes. In proportion to the notified priority, the VM scheduler adjusts the priorities of VMs. Since the VM scheduler considers only the highest priority in each VM, the other processes in the same VM can take too much CPU time. In task-grain scheduling [7], on the other hand, each guest OS notifies the hypervisor, the L4-embedded microkernel, of the priorities of all processes. Whenever a context switch occurs, the hypervisor schedules the guest OS running the process with the globally highest priority at that time. For regular VMMs such as Xen, switching VMs so frequently causes large overheads. Unlike the Monarch scheduler, these scheduling mechanisms require the modification of guest OSes.

In task-aware VM scheduling [20], on the other hand, the VM scheduler preferentially schedules VMs that execute I/O-bound processes without modifying guest OSes. The scheduler detects I/O-bound processes in VMs by using the same technique as Antfarm [12] and other gray-box knowledge. When a network packet arrives to the VMM, the

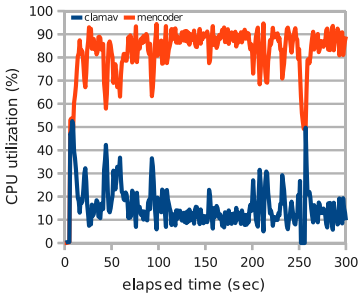


(a) Default scheduling

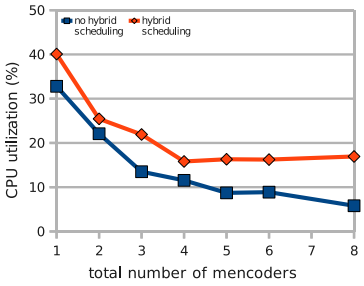


(b) Priority scheduling

Figure 9. System-wide priority scheduling for MEncoder and ClamAV.



(a) Hybrid scheduling



(b) Impact of MEncoder

Figure 10. The effects of hybrid scheduling with priority scheduling.

scheduler immediately schedules the VM where a process to receive the packet exists. This mechanism is for better VM scheduling, not for process scheduling.

A system-wide process scheduler can be also implemented using a technique similar to coordinated scheduling, which is used in distributed systems [21]. A local scheduler running in each VM obtains information on processes in the whole system by communicating with the other VMs. Then each local scheduler controls the execution of processes in the same VM. Since legacy OSes cannot be modified, local schedulers are often implemented as processes, using techniques for user-level scheduling [22], [23]. However, if the attackers compromise a local scheduler, they may easily perform DoS attacks by telling a lie to the other VMs. In addition, the process time recorded inside VMs may be inaccurate, as described in Section III-A.

Virtual machine introspection (VMI) is widely used for inspecting guest OSes from the VMM. Livewire [8] enables executing intrusion detection systems in the outside of a VM. IntroVirt [24] enables obtaining more information such as file contents from the VMM by executing kernel functions in a guest OS. To prevent kernel data from being modified by the execution of kernel functions, it uses checkpoint and rollback. VMwatcher [25] detects malwares by comparing information obtained from a guest OS with that obtained from the VMM. Lares [26] executes security applications in another VM by inserting hooks into the code of a guest OS from the VMM. While these systems only inspect guest OSes from the VMM, the Monarch scheduler modifies them as well.

XenAccess [27] is a library for inspecting guest OSes including Windows. The Monarch scheduler cannot use it in two reasons. First, it is a library available only in domain 0 of Xen. In general, domain 0 is a reasonable place to inspect domain Us. However, it is not suitable for a system-wide process scheduler because of its overheads. Domain 0 has to map memory pages into its address space to access the memory of domain Us. Second, XenAccess does not support to modify guest OSes.

Direct kernel object manipulation (DKOM) [28] is a technique that manipulates data in guest OSes by directly modifying the kernel memory. This technique has been often used for attacks. We use this technique for changing the scheduling behavior of guest OSes from the VMM. It was challenging to change the behavior of guest OSes by using only the DKOM technique because this technique can change it only indirectly through the modification of kernel data.

VII. Conclusion

In this paper, we proposed the Monarch scheduler for secure system-wide process scheduling across VMs. The Monarch scheduler is running in the VMM and monitors the execution of processes in all VMs, based on the switches of

their virtual address spaces. Then it changes the scheduling behavior of guest OSes by consistently manipulating run queues and process states without modifying guest OSes. To mitigate DoS attacks, its hybrid scheduling takes trade-off between system-wide scheduling and performance isolation among VMs. We showed that the Monarch scheduler could achieve useful scheduling policies and the overheads were small. According to our inspection of the Linux kernels, the Monarch scheduler is not affected by kernel updates in most cases.

One of the future work is supporting Windows guest OSes completely. In the current implementation, the Monarch scheduler can manipulate Windows processes in the run queues but cannot suspend ones in the running or blocked state. We need different techniques from ones used for Linux. Another direction is developing various system-wide scheduling policies using the Monarch scheduler. Possible scheduling policies also depends on scheduling algorithms in guest OSes.

ACKNOWLEDGEMENTS

This research was supported in part by JST, CREST.

REFERENCES

[1] L. Eggert and J. Touch, "Idletime Scheduling with Preemption Interval," in *Proc. Symp. Operating System Principles*, 2005, pp. 249–262.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proc. Symp. Operating Systems Principles*, 2003, pp. 164–177.

[3] ClamWin Team, "ClamWin Free Antivirus," http://www.clamwin.com/.

[4] Google, Inc., "Google Desktop," http://desktop.google.com/.

[5] Microsoft Corp., "SQL Server."

[6] D. Kim, H. Kim, M. Jeon, E. Seo, and J. Lee, "Guest-aware Priority-based Virtual Machine Scheduling for Highly Consolidated Server," in *Proc. Int. Euro-Par Conf. Parallel Processing*, 2008, pp. 285–294.

[7] Y. Kinebuchi, M. Sugaya, S. Oikawa, and T. Nakajima, "Task Grain Scheduling for Hypervisor-Based Embedded System," in *Proc. Int. Conf. High Performance Computing and Communications*, 2008.

[8] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[9] J. N. Petroni and M. Hicks, "Automated Detection of Persistent Kernel Control-flow Attacks," in *Proc. Conf. Computer and Communications Security*, 2007.

[10] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh, "Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proc. USENIX Security Symp.*, 2004.

[11] Trusted Computing Group, http://www.trustedcomputinggroup.org/.

[12] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *Proc. USENIX Annual Technical Conf.*, 2006, pp. 1–14.

[13] DWARF Standards Committee, "The DWARF Debugging Standard," http://dwarfstd.org/.

[14] J. Kneschke, "lighttpd," http://www.lighttpd.net/.

[15] The Apache Software Foundation, "Apache HTTP Server Benchmarking Tool," http://httpd.apache.org/.

[16] M. Hirabayashi, "Hyper Estraier: a Full-text Search System for Communities," http://hyperestraier.sourceforge.net/.

[17] J. Zhang and M. Wong, "Database Test 3," http://osdldbt.sourceforge.net/.

[18] Sourcefire, Inc., "Clam AntiVirus," http://www.clamav.net/.

[19] MPlayer Team, "MPlayer – The Movie Player," http://www.mplayerhq.hu/.

[20] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware Virtual Machine Scheduling for I/O Performance," in *Proc. Int. Conf. Virtual Execution Environments*, 2009, pp. 101–110.

[21] J. Ousterhout, "Scheduling Techniques for Concurrent Systems," in *Proc. Int. Conf. Distributed Computing Systems*, 1982, pp. 22–30.

[22] T. Newhouse and J. Pasquale, "A User-Level Framework for Scheduling within Service Execution Environments," in *Proc. Int. Conf. Services Computing*, 2004, pp. 311–318.

[23] T. Newhouse and J. Pasquale, "ALPS: An Application-Level Proportional-share Scheduler," in *Proc. Int. Symp. High Performance Distributed Computing*, 2006, pp. 279–290.

[24] A. Joshi, S. King, G. Dunlap, and P. Chen, "Detecting Past and Present Intrusions through Vulnerability-specific Predicates," in *Proc. Symp. Operating Systems Principles*, 2005, pp. 91–104.

[25] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection through VMM-based "Out-of-the-box" Semantic View Reconstruction," in *Proc. Conf. Computer and Communications Security*, 2007, pp. 128–138.

[26] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *Proc. Symp. Security and Privacy*, 2008, pp. 233–247.

[27] B. Payne, M. Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Proc. Annual Conf. Computer Security Applications*, 2007, pp. 385–397.

[28] J. Butler, "DKOM (Direct Kernel Object Manipulation)," Black Hat Windows Security, 2004.