URS: User-Based Resource Scheduling for Multi-User Surface Computing Systems

Jonggyu Park and Young Ik Eom

***Abstract*—Multi-user surface computing systems are promising for the next generation consumer electronics devices due to their convenience and excellent usability. However, conventional resource scheduling schemes can cause severe performance issues in surface computing systems, especially when they are adopted in multi-user environments, because they do not consider the characteristics of multi-user surface computing systems. In this paper, we propose an efficient user-based resource scheduling scheme for multi-user surface computing systems, called URS. URS provides three different features to effectively support multi-user surface computing systems. First, URS distributes system resources to users, according to their real-world priorities, rather than processes or tasks. Second, URS provides performance isolation among multiple users to prevent resource monopoly by a single user. Finally, URS prioritizes a foreground application of each user via retaining pages used by the application in the page cache, in order to enhance multi-user experience. Our experimental results confirm that URS effectively allocates system resources to multiple users by providing the aforementioned three features.**

***Index Terms*—Multi-user surface computing, resource scheduling, page cache management.**

I. Introduction

With technical advances in touch interfaces, consumer electronics (CE) devices such as smartphones, tablets, and display devices [1] are adopting surface computing systems, in which users can interact with the devices using touch interfaces. This touch technology has dramatically changed human life, by eliminating the requirement of some traditional hardware interfaces, such as a button or a mouse. Moreover, enlargement of display size in recent years [2], [3] has brought a possibility of multi-user surface computing systems where multiple users simultaneously run their individual applications on a single device using touch interfaces [2], [4], [5]. In such systems, multiple users collectively share a single device, thereby improving the efficiency and productivity of collaborative work, which personal computing systems cannot simply achieve. For example, multiple users can easily exchange data or applications each other on a single device without network support, eliminating unnecessary communication overheads. In addition, they can collaboratively operate applications for a special purpose using a single large display. Hence, along with personal computing systems, multi-user surface computing systems are expected to be widely utilized for various purposes including education, entertainment, and collaborative work in the future [3], [6], [7].

In multi-user surface computing systems [2], [4], [5], [6], [7], there are several system requirements that should be met in order to provide stable overall performance and enhance multi-user experience. First, the systems should basically provide performance fairness among users and also be able to prioritize a particular user. We refer to this feature as user-based scheduling in this paper. In multi-user environments, users with identical priority should also have the same amount of resources in order to enhance multi-user experience. Additionally, when a user has a higher real-world priority, the user should take more system resources than other users. However, since conventional scheduling schemes [8], [9], [10] distribute system resources to processes or tasks, rather than users, it is possible for users who run more applications to have more amount of resources than other users even though they have identical priority. Such performance unfairness diminishes multi-user experience. Moreover, conventional systems are unable to prioritize applications of a user who has a higher real-world priority. For example, modern Linux systems distribute resources based on processes or sessions [8], [11]. For this reason, they cannot provide performance fairness among users, and are unable to selectively prioritize applications of a particular user in the system level. To sum up, conventional systems do not provide user-based scheduling, degrading the multi-user experience.

Second, a system for multi-user surface computing should provide performance isolation among users. To enhance multi-user experience, the system should minimize the performance interference among users. Additionally, the system should guarantee that each user can run his/her application even under high resource utilization. However, in conventional systems, since the amount of resources provided to each user is determined by the number of his/her applications, a malicious user might have the ability to degrade the performance of the entire system, thereby contaminating the performance of other users' processes. In the worst case, a single malicious user can halt the entire system by simultaneously running a large number of applications. This type of

Manuscript received January 23, 2019; revised May 7, 2019; accepted June 17, 2019. Date of publication June 24, 2019; date of current version July 24, 2019. This work was supported by the Institute for Information and Communications Technology Promotion (IITP) grant funded by the Korea Government (MSIT) (IITP-2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-Users on Large Displays). *(Corresponding author: Young Ik Eom.)*

The authors are with College of Software, Sungkyunkwan University, Suwon 2066, South Korea (e-mail: jonggyu@skku.edu; yieom@skku.edu).

Digital Object Identifier 10.1109/TCE.2019.2924480

performance interference can reduce the usability of multi-user systems.

Third, multi-user systems should be able to prioritize each user's foreground application. A foreground application is critical to user experience because users usually wait for a response from the foreground applications [12], [13], [14]. However, in contrast to single user systems like a mobile device which has just one foreground application [12], [13], [14], there are multiple foreground applications in multi-user systems, one for each user [2], [4], [5]. Therefore, multi-user systems should assure that multiple foreground applications react to user inputs without noticeable delays.

In this paper, we propose an efficient user-based resource scheduling scheme for multi-user surface computing systems, called URS, which satisfies the aforementioned three requirements. URS creates an isolated resource group for each user, called a user group, and distributes system resources based on these user groups. Since a user group is isolated from other user groups, our scheme can achieve performance isolation among the users. Additionally, URS can achieve user-based scheduling by assigning priorities to each user group. For example, URS can provide performance fairness among users by assigning the same priority to all the user groups. Furthermore, it can prioritize a crucial user's applications by assigning them a higher priority. The previous version of URS [15] manages only CPU and I/O resources. However, this paper also handles the page cache to prioritize multiple foreground applications. Whenever a foreground application allocates a new page in the page cache, URS denotes that the page is being used by foreground applications. When pages in the page cache need to be evicted, URS provides additional opportunities for the marked pages to stay in the page cache, thereby prioritizing multiple foreground applications in a multi-user system.

The remainder of this paper is organized as follows. In Section II, we explain the background of this work, such as the description of multi-user surface computing systems and conventional resource scheduling schemes. A detailed explanation of URS is given in Section III. Experimental results follow in Section IV, and related work is summarized in Section V, which compares our scheme with other schemes such as Fair share scheduler and Cgroups. Finally, Section VI concludes this paper with future work.

II. Background

This section introduces the concept and description of multi-user surface computing systems, and explains how conventional systems manage the system resources, such as CPU, I/O, and the page cache.

A. Multi-User Surface Computing Systems

Conventional CE devices rely on hardware interfaces such as a button or a mouse for user interactivity. However, touch technology entirely changes the way that users interact with the devices. Touch technology allows users to communicate with devices using their fingers, eliminating the need for hardware interfaces. Additionally, enlargement of display size increases the number of users who can simultaneously run individual applications using a single device. Therefore, such multi-user surface computing systems are expected to be used in various fields such as smart homes and smart offices, and show promise for the next generation CE devices [16].

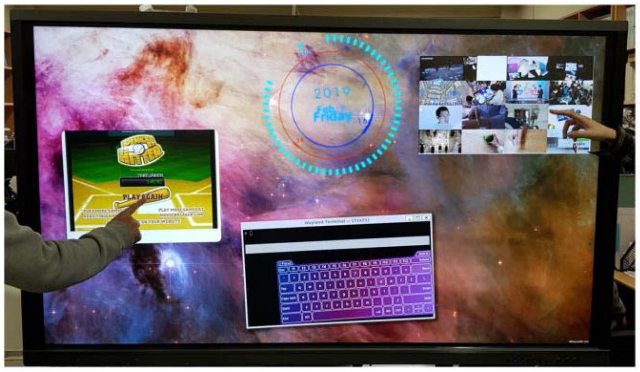

Fig. 1. An example of multi-user surface computing systems.

Multi-user surface computing systems are different from conventional single-user systems for the following reasons. First, the system should distribute system resources to several users, according to their real-world priorities, in order to enhance multi-user experience. Second, since multiple users run their applications simultaneously, the system should address performance interference among the users, so as to enhance the usability of the systems. Third, there can be multiple foreground applications, which affects the quality of user experiences, unlike in single-user systems. Hence, multi-user systems should prioritize the multiple foreground applications to improve the quality of multi-user experience.

B. Conventional CPU and I/O Scheduling

Current Linux systems utilize CFS (Completely Fair Scheduler) [8] to equitably distribute CPU resources to processes. Moreover, current Linux systems provide group scheduling, which creates a group of processes, called a task group, and schedules CPU resources for the task groups rather than processes. By default, Linux systems automatically make a group of processes from each session to prevent a monopoly by a single session [8]. CFS manages per-core runqueue which contains scheduling entities such as processes or task groups. CFS tags a scheduling entity with *'vruntime'*, which indicates how long the scheduling entity has been run, and allocates CPU resources to the scheduling entity with lowest *'vruntime'* when the core is available.

For scheduling I/O resources, current Linux systems provide several options, such as CFQ (Completely Fair Queueing) [11], NOOP, Deadline, etc. Among these options, this paper focuses on CFQ since only CFQ considers fairness. CFQ manages per-process I/O queues to provide fairness among the processes. Each queue has a specific time slice, called *'vdisktime'* that is decided by its *'weight'*. CFQ handles requests in each queue within the *'vdisktime'* and moves to the next queue in a round-robin way after the *'vdisktime'* is finished.

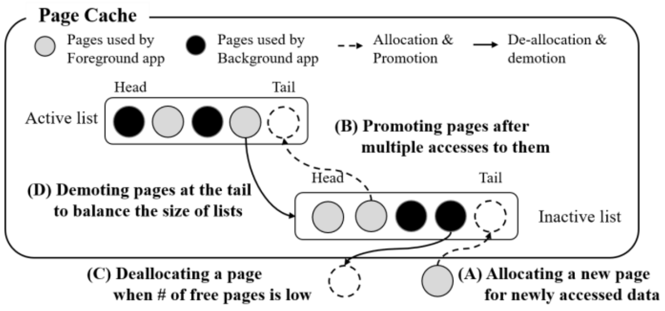
Page Cache
Pages used by Foreground app
Pages used by Background app
Allocation & Promotion
De-allocation & demotion
Head
Tail
Active list
(B) Promoting pages after multiple accesses to them
(D) Demoting pages at the tail to balance the size of lists
Head
Tail
Inactive list
(C) Deallocating a page when # of free pages is low
(A) Allocating a new page for newly accessed data

Fig. 2. The conventional page cache management.

Like this, conventional resource scheduling schemes distribute CPU and I/O resources only based on processes or sessions, without considering users. These conventional scheduling schemes can incur various problems in multi-user surface computing systems. First, a user that runs more processes or sessions will use more system resources than other users, resulting in performance unfairness among the users. Second, when a malicious user run a large number of processes, the entire system can halt because the malicious user monopolizes almost all the system resources. Hence, conventional scheduling schemes has a lack of consideration for multi-user surface computing systems. To solve these problems, a new resource scheduling scheme for multi-user surface computing systems need to be developed.

C. *Conventional Page Cache Management*

The page cache is a specialized space residing in main memory to bridge the performance gap between main memory and secondary storage. Since secondary storage is much slower than main memory, issuing I/Os to the storage can severely increase the latencies. To compensate for the low performance of the storage, the page cache retains frequently or recently accessed data in main memory. When the upcoming I/Os access the cached pages, the I/Os are processed in the page cache without data transmission from/to the storage.

As depicted in Fig. 2, the modern page cache maintains two different LRU queues, called the active list and the inactive list, to efficiently manage pages in the page cache [17]. (A) When a new page is allocated in the page cache, it is first inserted into the inactive list. (B) After the page is accessed several times (by default, more than twice), it is promoted to the active list. The eviction process is opposite to the allocation process. (C) When the system has insufficient free pages in main memory, the page cache management de-allocates pages at the tail of the inactive list to reclaim free pages. Additionally, (D) it demotes pages at the tail of the active list to the inactive list, in order to balance the size of the lists.

The conventional page cache management does not distinguish pages used by a foreground application from those used by background applications. Thus, the latency of a foreground application can be increased, diminishing the quality of user experience. In multi-user surface computing systems, there can be multiple foreground applications, and therefore, a new page cache management scheme that prioritizes multiple foreground applications is necessary to enhance multi-user experiences.

III. Design and Implementation

As mentioned in Section II, conventional resource scheduling, including CPU, I/O, and the page cache, has a lack of consideration for the features of multi-user surface computing systems. Therefore, conventional resource scheduling schemes can cause severe degradation of user experience in multi-user surface computing systems.

In this paper, we present a user-based resource scheduling scheme for multi-user surface computing systems, called URS, that improves multi-user experience by providing performance fairness and isolation, and prioritizing each user's foreground application. For user-based scheduling and performance isolation among users, URS creates an individual resource group for each user, called a user group, and inserts each application to its corresponding user group. To prioritize foreground applications, URS detects pages that are used by the foreground applications, and tries to keep the pages in the page cache longer. We implement the user group using the Cgroups implementation, which controls resources based on groups and provides isolation among the groups. The details of Cgroups will be discussed in Section V.

A. *Constructing User Groups for CPU and I/O Resources*

URS creates an individual user group per user for CPU and I/O resources, called "User Task Group" and "User CFQ Group", respectively. To create a user group for each user, URS needs to distinguish the ownership of applications. There have been several research results [18], [19], [20] related to how to pair applications and their owners. Since the scope of our paper does not include this subject, we suppose that users log in to the system before they use it, and thus the current version of URS simply utilizes the *'uid'* (user id) of applications to define the owners of applications.

As described in Fig. 3, when a user logs in, URS checks if a user group with the *'uid'* exists in the user group list by hooking system calls. If it exists, URS inserts the current process of the user into the user group. Otherwise, URS creates a new user group linked to the user group list and inserts the current (log-in) process of the user into the user group. A child process forked by a parent process belongs to the same user group because the child process copies the group information of the parent process. Therefore, if URS inserts a user's current process into the user group on log-in, all the processes that the user executes, belong to the same user group.

B. *CPU Scheduling for Performance Fairness and Isolation Among Users*

In CPU scheduling, URS constructs three layers of task groups using the method explained in the previous subsection. As described in Fig. 4, the first layer is the root task group, of which runqueue contains all the user task groups as its scheduling entities. The second layer is the user task group, which URS provides to each user. Under user task groups, there exist session task groups that the user owns. We maintain the session task groups to benefit from the conventional CPU scheduling. Finally, a runqueue of each session task group contains actual processes created on the session.

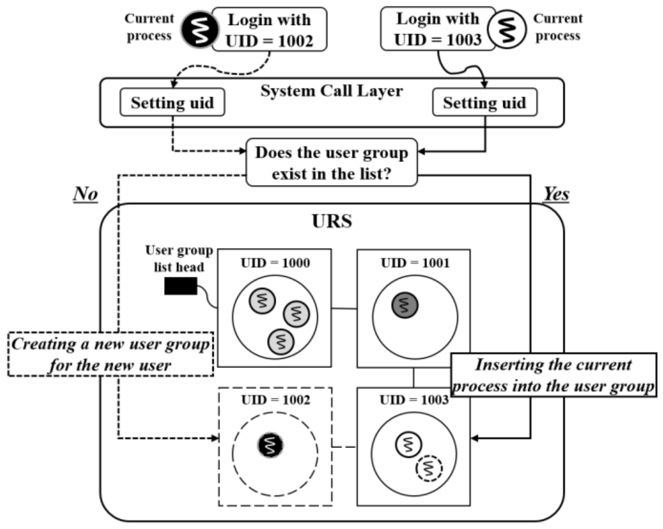

Fig. 3. How to construct user groups.

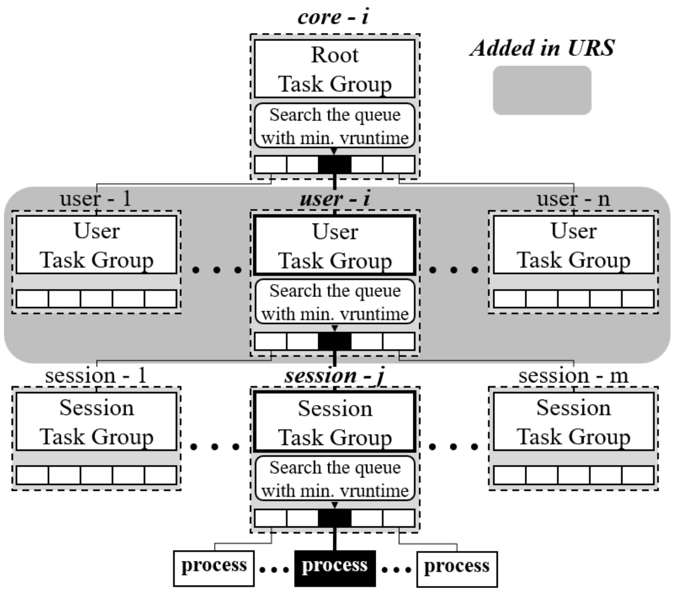

Fig. 4. CPU scheduling of URS for performance fairness and isolation among users.

When URS schedules CPU resources, it first chooses a user task group with the lowest *'vruntime'*. And then, URS chooses a session task group with the lowest *'vruntime'* in the user task group. Finally, URS allocates CPU resources to the process with the lowest *'vruntime'* in the session task group. In our CPU scheduling scheme, each user task group is an isolated scheduling entity from the other user task groups, and the system allocates CPU resources to each user task group, depending on its specified *'share'*. (The higher the *'share'* value is, the more CPU resources are allocated because the *'vruntime'* increases slowly.) Therefore, CPU scheduling of URS provides CPU performance isolation among users by creating a task group for each user. Additionally, it can provide CPU performance fairness by assigning the same *'share'* value

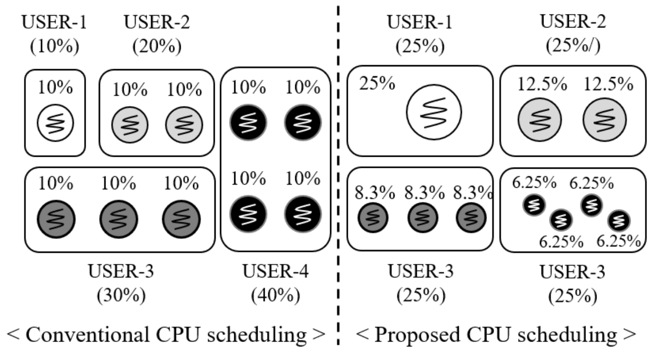

Fig. 5. Comparison between the conventional and proposed CPU scheduling.

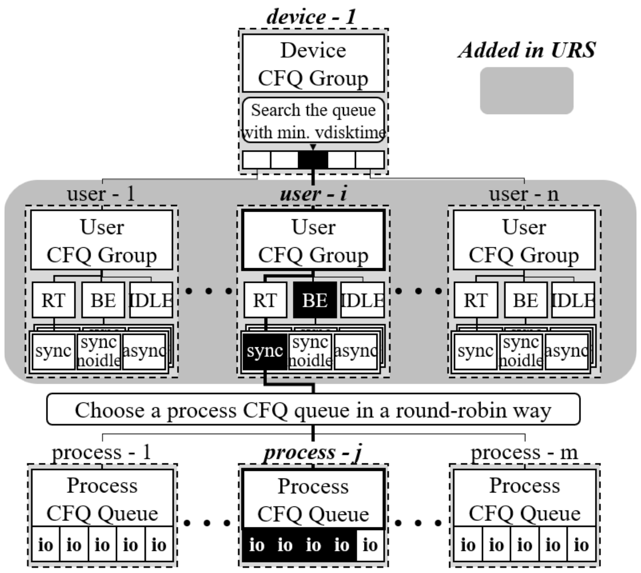

Fig. 6. I/O scheduling of URS for performance fairness and isolation among users.

to all the user task groups, or it can prioritize applications of a particular user by assigning a higher *'share'* value to the user.

For example, let us assume that there are four users who run 1-4 applications, each of which is running on an individual session, as shown in Fig. 5. In the conventional system, CPU resources are distributed based on sessions. Therefore, the system allocates quadruple the amount of resources to the user who runs four applications, compared to the user who runs only a single application. However, in case of the proposed scheme, all users have their own user group above the session task groups. Therefore, the system can allocate the same amount of CPU resources to each user regardless of the number of their applications, when all users have the same *'share'* value.

C. I/O Scheduling for Performance Fairness and Isolation Among Users

I/O scheduling of URS is similar to the CPU scheduling. In conventional I/O scheduling, each process has its own CFQ queue to ensure fairness among the processes. On the other hand, the proposed scheme additionally creates a user CFQ group for each user above the process CFQ queue, as described in Fig. 6. In a user CFQ group, each process queue is mapped to a specific service tree, depending on the priority (RT, BE, IDLE) and I/O type (Sync, Async), where RT and BE stands for real-time and best effort, respectively. For I/O scheduling, the system assigns *'vdisktime'* to each user CFQ group. The *'vdisktime'* denotes how long a group has used I/O resources by considering its *'weight'* (The higher the *'weight'* value is, the more I/O resources are allocated because the *'vdisktime'* slowly increases). Since I/O scheduling of URS allocates a *'weight'* value to each user CFQ group, the system can control I/O resources based on users. In other words, URS can provide I/O performance fairness by assigning the same *'weight'* value to all the user CFQ groups and prioritize applications of a particular user by assigning a higher *'weight'* value to the user CFQ group. Additionally, since each user CFQ group is isolated from other user CFQ groups with regard to I/O resources, as in CPU scheduling, the proposed scheme also provides I/O performance isolation among the users. The current version of URS supports CFQ I/O scheduler first because of its importance, and we will extend our scheme to support various I/O schedulers, such as Deadline and NOOP I/O scheduler in the future.

D. Page Cache Management for Prioritizing Multiple Foreground Applications

As discussed in Section II, the current page cache management is oblivious of the existence of foreground applications. This can seriously degrade the quality of multi-user experience because there are multiple foreground applications in a multi-user surface computing system. To solve this problem, URS distinguishes pages used by foreground applications, and tries to keep these pages longer in the page cache. The way to distinguish foreground applications from background ones can differ depending on the types of platforms. For example, in the Android platform, the system can obtain the *'pid'* (process id) of a foreground application by monitoring the activity stacks [13]. In multi-user surface computing systems, it can be done by analyzing the interactivity between users and applications. In this paper, since detecting foreground applications is not in the scope of this paper, we simply distinguish foreground applications by providing hints from the application layer.

As described in Fig. 7, in our scheme, we add two variables into the page structure, named *'usedbyfg'* and *'opportunity'*. *'usedbyfg'* denotes whether or not the page is used by a foreground application. *'opportunity'* shows the number of possible deferrals of the page's demotion. When an application issues I/Os and the kernel allocates a new page for the I/O in the page cache, URS checks whether the application is a foreground application or not. (A) If it is, URS sets *'usedbyfg'* of the new page to true and assigns an initial value to *'opportunity'* of the new page. The initial value for *'opportunity'* differs depending on how important the foreground applications are, compared to the background ones. In our experiments, we define the initial value as 10. (B) If the page is not used by foreground applications, URS sets *'usedbyfg'* to false and assigns zero to *'opportunity'*.

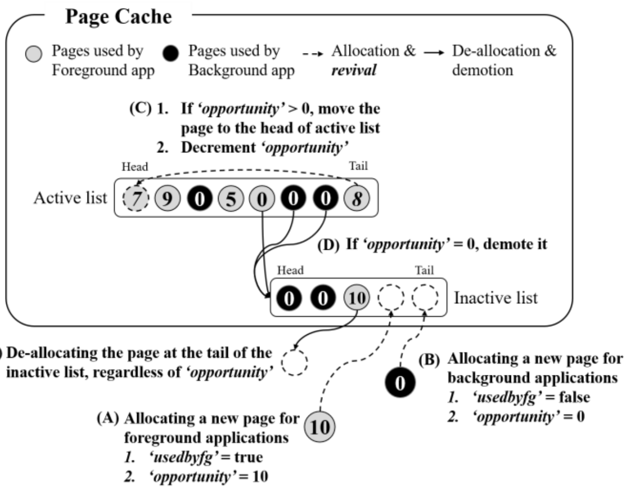
Page Cache
Pages used by Foreground app
Pages used by Background app
Allocation & ***revival***
De-allocation & demotion
(C) 1. If *'opportunity'* > 0, move the page to the head of active list
2. Decrement *'opportunity'*
Head
Tail
Active list
7
9
0
5
0
0
0
8
(D) If *'opportunity'* = 0, demote it
Head
Tail
0
0
10
Inactive list
(E) De-allocating the page at the tail of the inactive list, regardless of *'opportunity'*
(B) Allocating a new page for background applications
1. *'usedbyfg'* = false
2. *'opportunity'* = 0
0
(A) Allocating a new page for foreground applications
1. *'usedbyfg'* = true
2. *'opportunity'* = 10
10

Fig. 7. Page cache management for prioritizing multiple foreground applications.

When URS needs to demote pages from the active list to the inactive list so as to balance the size of the lists, URS checks the *'opportunity'* value of the page at the tail of the active list to confirm whether it is greater than zero or not. (C) If it is greater than zero, URS moves the page to the head of the active list, in order to give the page more opportunities to remain in the active list, while decrementing the *'opportunity'* value. (D) If the value of *'opportunity'* is zero, URS demotes the page to the inactive list. By so doing, URS retains pages being used by foreground applications in the page cache longer. As a result, URS prioritizes a foreground application of each user.

URS assigns the *'opportunity'* variable of a page with the initial value (10 in our experiments) again whenever the page is accessed by foreground applications. We do not recommend setting the initial value to infinite since it can decrease the overall hit ratio of the page cache by retaining useless data. (E) URS does not give opportunities to pages in the inactive list, because delaying deallocation of the pages in the inactive list can increase the latency of page allocation and cause a problem of Out-of-Memory.

IV. EVALUATION

We implemented URS in Linux kernel version 4.11, by modifying the conventional scheduling and management schemes for CPU, I/O, and the page cache. To quantitatively evaluate URS, we performed experiments on a single computer equipped with 3.4GHz quad-core CPU, 8GB DRAM, and 120GB SSD.

To verify user-based scheduling of URS, we make four users run 1, 2, 3, and 4 identical workloads, respectively, in two different real-world situations. In the first situation, all the users have the same priorities to verify performance fairness among users (denoted as 'fair' in Fig. 8-12). In the second situation, we give user2 twice as much priority as the other users, to check if URS can prioritize a particular user when the user has a higher priority (denoted as 'user2' in Fig. 8-12). From the two experiments, we confirm whether the URS distributes resources to users based on their real-world priorities

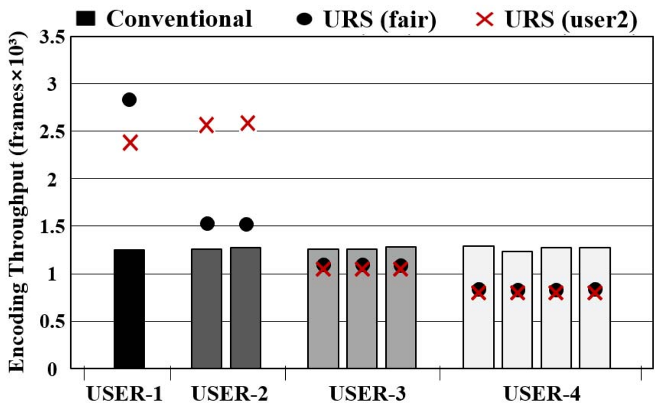

Fig. 8. User-based scheduling experiments for CPU scheduling of URS.

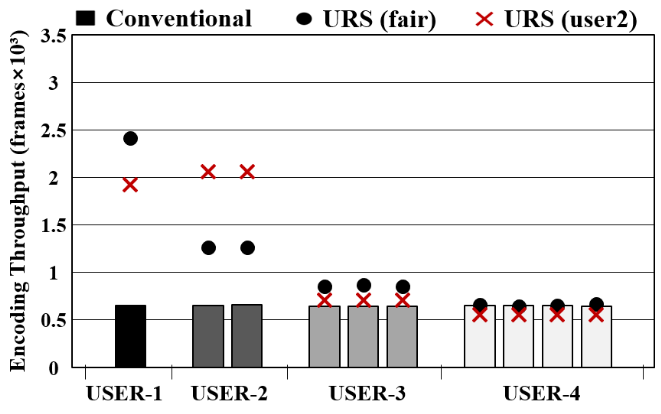

Fig. 9. Performance isolation experiments for CPU scheduling of URS.

(user-based scheduling) or not. Note that we do not include the results of conventional scheduling in the second situation since it does not have the ability to prioritize a particular user.

To verify performance isolation among multiple users, we also performed the same experiments, while adding a malicious user who runs ten stress-ng benchmarks to overload the system. In these experiments, we measured how much the performance of legitimate users could be degraded by a single malicious user. We did not include the performance results of the stress-ng benchmarks, because the benchmarks were used solely for hindering the applications of the other users (Also, the malicious user is not shown in the figures).

To confirm the effectiveness of our page cache management, we additionally conducted an experiment in which each of four users runs one foreground and one background applications. In this experiment, we measured the performance of foreground applications to check if URS can prioritize foreground applications over background applications. All the experiments in this paper were conducted under both conventional schemes (Linux schedulers and page cache) and URS.

A. CPU Scheduling of URS

In CPU experiments, users run MEncoder, which is a video encoding tool. We decided to use MEncoder because video applications are commonly used on multi-user surface computing devices in real-world situations. Using MEncoder, each application encoded 100MB video data. Fig. 8 shows the effectiveness of CPU scheduling. First, let us see the results of the first situation where all users have the same real-world priority (*'fair'* in Fig. 8). Since the conventional scheduling scheme provides fairness among the sessions (or processes), and each application in the experiments runs on a different session, the more applications a user runs, the more CPU resources the user takes. As a result, when we calculate the aggregate throughput of processes per user, the conventional scheme allocates 2.02, 3.04, and 4.06 times more CPU resources to USER-2, USER-3, and USER-4, respectively, compared to USER-1. Therefore, the conventional system cannot provide performance fairness among multiple users.

On the other hand, URS fairly allocates CPU resources to all users by assigning the same *'share'* value to all the user task groups. As a result, the proposed scheme allocates a similar amount of CPU resources to all the users, with 18% or less difference. When USER-2 has twice the priority of the other users (*'user2'* in Fig. 8), URS assigns twice the *'share'* value of the other user task group to the user task group for USER-2. Eventually, USER-2 shows approximately two times higher aggregate encoding throughput than the other users. Therefore, URS provides user-based scheduling, which distributes resources based on the real-world priorities of users. Note that although all users are equally affected by the increase of USER-2's priority, the performance of USER-1 appears to be more degraded than the other users in Fig. 8 and Fig. 9. This is due to the fact that URS also provides performance fairness among the processes of each individual user. In the cases of USER-3 and USER-4, the performance degradation by the increase of USER-2's priority is distributed to multiple processes unlike the case of USER-1.

In the isolation experiments, as shown in Fig. 9, the proposed scheme shows 1.56 times higher aggregate throughput than the conventional scheme, when we calculate the aggregate throughput of the processes owned by normal users excluding the malicious user. In the conventional scheme, the malicious user degrades the overall system performance due to the lack of CPU performance isolation among the users. On the other hand, URS creates a user task group for each user to isolate the resources for each user from other users, and thus the interference of the malicious user can be minimized.

B. I/O Scheduling of URS

To evaluate the I/O scheduling of URS, we performed FIO read benchmark with O_DIRECT option that bypasses the page cache. The results of I/O experiments are similar to those of CPU experiments. In the first situation of the user-based scheduling experiments (*'fair'* in Fig. 10), the conventional scheme allocates 2.11, 3.11, and 4.15 times more I/O resources to USER-2, USER-3, and USER-4, respectively, compared to USER-1, because it allocates I/O resources based on processes.

On the other hand, the proposed scheme allocates a similar amount of I/O resources to all the users, with 15% or less difference by assigning the same priority to all the user CFQ groups. In the second situation (*'user2'* in Fig. 10), USER-2

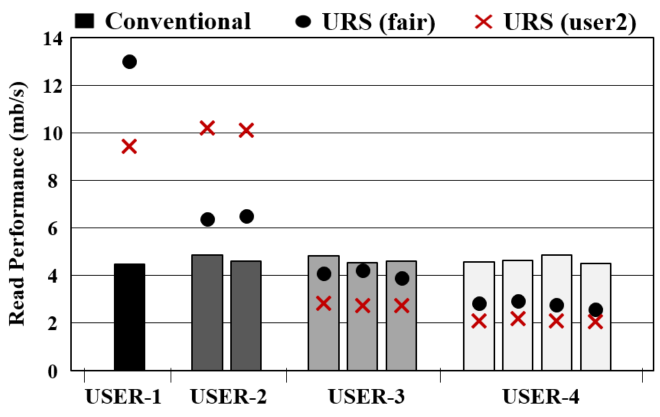

Fig. 10. User-based scheduling experiments for I/O scheduling of URS.

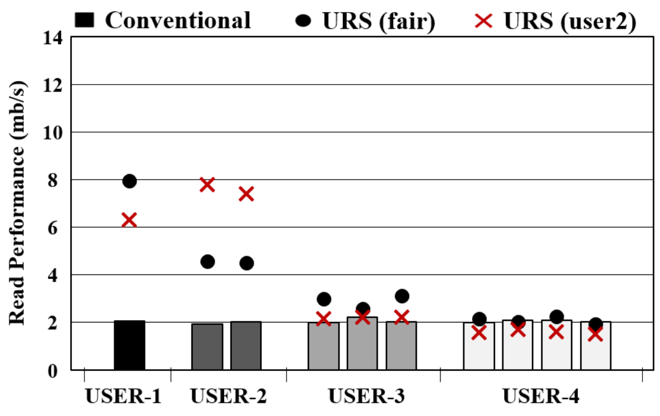

Fig. 11. Performance isolation experiments for I/O scheduling of URS.

shows around two times higher read performance in total than the other users because URS gives two times higher *'weight'* value to the corresponding user CFQ group than the other CFQ user groups.

On the isolation experiments in Fig. 11, the proposed scheme shows 1.66 times higher aggregate throughput than the conventional scheme. As with the CPU scheduling experiments, URS isolates each user from other users in terms of I/O performance by providing users with individual user CFQ groups. Accordingly, URS diminishes the influence of the malicious user on the normal users. As a result, URS provides user-based resource scheduling and performance isolation among multiple users for I/O resources, as well as CPU resources.

C. Page Cache Management of URS

To verify the page cache management of URS, we simulate a situation where four users run one foreground and one background application each. For both of the foreground and background applications, each user runs two FIO benchmarks, each of which reads a 1GB file with a request size of 128KB. To inform whether an application runs in the foreground or not, we explicitly set the different nice value to the foreground application. Also, we repeatedly run the applications for their data to be cached in the page cache layer.

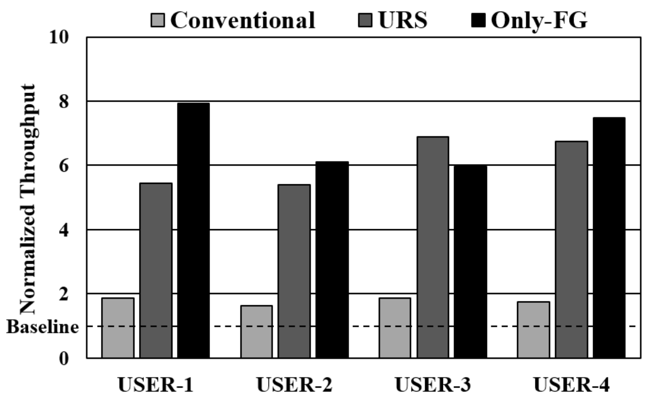

Fig. 12. FIO-read experiments for verifying the page cache management of URS.

The experimental results in Fig. 12 are normalized to the baseline where applications run with bypassing the page cache. In the baseline, all data should be read from the secondary storage. Only-FG in Fig. 12 shows the performance of the best case when only foreground applications are running without background ones. In Fig. 12, the conventional scheme shows 1.78 times higher average throughput than the baseline, whereas URS shows 6.12 times higher average throughput than the baseline. The conventional scheme does not distinguish the foreground applications, and thus their data are frequently deallocated from the page cache because of background applications. However, URS differentiates data of foreground applications and prioritizes them by giving more opportunities to stay in the page cache. Eventually, the average throughput of URS is only 11% less than that of Only-FG.

V. Related Work

Fair share scheduler [21] brings up the problem of scheduling resources based on processes and tries to equally distribute CPU resources to users or groups. This work presents the concept of '*share*' which indicates the entitlement to take resources. Fair share scheduler assigns the same amount of *'share'* to each user and distributes resources based on the *'share'* values, resulting in performance fairness among users or groups. However, the fair share scheduler only focuses on the users' fairness without prioritizing the users according to the real-world priorities. Additionally, it does not accelerate foreground applications which are critical to the user experience of multi-user surface computing systems.

Another previous work related to the user-based resource scheduling is Cgroups. Cgroups is a user-space API to control and limit system resources, such as CPU, memory, and I/O, via creating a resource group. By making a resource group per user in the user-space, Cgroups can distribute system resources based on users and isolate users from each other. However, since Cgroups works in the user-space, it should keep track of the ownership information of all applications by continuously inquiring it to the kernel layer. Besides, Cgroups cannot control pages for the page cache, and thus it cannot prioritize foreground applications in the page cache layer.

FastTrack [12] mentions the importance of a foreground application regarding user experience. It improves the I/O performance of a foreground application by prioritizing data being used by the foreground application over other data in the page cache layer. However, FastTrack only deals with the mobile environment where only one foreground application is running. Therefore, FastTrack cannot be applied to multi-user surface computing systems where multiple foreground applications are running.

Finally, VDI (virtual desktop infrastructure) is similar to multi-user surface computing in that multiple users share a single device. One might think scheduling schemes in VDI can be directly adopted in multi-user surface computing systems. However, in VDI, each desktop instance is controlled and isolated from each other using virtualization techniques, unlike multi-user surface computing systems. For this reason, the host CPU and I/O schedulers in VDI do not consider the performance fairness or isolation among users. Therefore, a new scheduling scheme is necessary to effectively support multi-user surface computing systems.

VI. Conclusion & Future Work

This paper presents a user-based scheduling scheme for multi-user surface computing systems, called URS. In contrast to the conventional schemes, URS creates a user group for each user that is an isolated scheduling entity derived from the Cgroups implementation and distributes system resources based on the user groups. URS controls the amount of CPU and I/O resources allocated to user groups by assigning a *'share'* value and a *'weight'* value to each user group, respectively. By doing so, URS reflects the real-world priorities of users in scheduling system resources and provides performance isolation among users. Additionally, URS prioritizes a foreground application of each user by keeping the pages used by it in the page cache longer. The experimental results confirm that URS isolates system resources among multiple users and distributes resources to the users based on the real-world priorities of the users. Also, the experiment shows that it prioritizes the foreground application of each user.

In the future, we will evaluate the effectiveness of URS with various real-world workloads on a real device for multi-user surface computing. In addition, we are planning to incorporate the prior work, which discerns the ownership of applications in real world, into URS.

References

[1] B. Lee, I. Hong, Y. Uhm, and S. Park, "The multi-touch system with high applicability using tri-axial coordinate infrared LEDs," *IEEE Trans. Consum. Electron.*, vol. 55, no. 4, pp. 2416–2424, Nov. 2009. doi: 10.1109/TCE.2009.5373818.

[2] J. Leigh *et al.*, "Scalable resolution display walls," *Proc. IEEE*, vol. 101, no. 1, pp. 115–129, Jan. 2013. doi: 10.1109/JPROC.2012.2191609.

[3] K. Wang, S. Lian, and Z. Liu, "An intelligent screen system for context-related scenery viewing in smart home," *IEEE Trans. Consum. Electron.*, vol. 61, no. 1, pp. 1–9, Feb. 2015. doi: 10.1109/TCE.2015.7064104.

[4] J. Kim, I. Kim, T. Kim, and Y. I. Eom, "NEMOSHELL demo: Windowing system for concurrent applications on multi-user interactive surfaces," in *Proc. ACM ITS*, Dresden, Germany, 2014, pp. 451–454.

[5] I. Kim, J. Kim, J. Park, and Y. I. Eom, "Software-based single-node multi-GPU systems for interactive display wall," *IEEE Trans. Consum. Electron.*, vol. 63, no. 2, pp. 101–108, May 2017. doi: 10.1109/TCE.2017.014822.

[6] P. Dillenbourg and M. Evans, "Interactive tabletops in education," *Int. J. Comput. Support. Collab. Learn.*, vol. 6, no. 4, pp. 491–514, Dec. 2011. doi: 10.1007/s11412-011-9127-7.

[7] Y. Ahn *et al.*, "A slim hybrid multi-touch tabletop interface with a high definition LED display and multiple cameras," in *Proc. IEEE ICCE*, Las Vegas, NV, USA, 2012, pp. 1–2.

[8] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The Linux scheduler: A decade of wasted cores," in *Proc. ACM EuroSys*, London, U.K., 2016, pp. 1–16.

[9] J. Wei, R. Ren, E. Juarez, and F. Pescador, "A Linux implementation of the energy-based fair queuing scheduling algorithm for battery-limited mobile systems," *IEEE Trans. Consum. Electron.*, vol. 60, no. 2, pp. 267–275, May 2014. doi: 10.1109/TCE.2014.6852003.

[10] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards achieving fairness in the Linux scheduler," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 34–43, Jul. 2008. doi: 10.1145/1400097.1400102.

[11] J. Axboe, "Linux block IO—Present and future," in *Proc. OLS*, Ottawa, ON, Canada, 2004, pp. 51–62.

[12] S. Kim, H. Kim, J. Hwang, J. Lee, and E. Seo, "An event-driven power management scheme for mobile consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 59, no. 1, pp. 259–266, Feb. 2013. doi: 10.1109/TCE.2013.6490268.

[13] S. S. Hahn, S. Lee, I. Yee, D. Ryu, and J. Kim, "FastTrack: Foreground app-aware I/O management for improving user experience of Android smartphones," in *Proc. USENIX ATC*, Boston, MA, USA, 2018, pp. 15–27.

[14] S.-H. Kim, J. Jeong, and J. Lee, "Selective memory deduplication for cost efficiency in mobile smart devices," *IEEE Trans. Consum. Electron.*, vol. 60, no. 2, pp. 276–284, May 2014. doi: 10.1109/TCE.2014.6852004.

[15] H. Kim, J. Park, and Y. I. Eom, "User-based resource scheduling for multi-user surface computing systems," in *Proc. IEEE ICCE*, Las Vegas, NV, USA, 2018, pp. 1–3.

[16] S. S. Intille, "Designing a home of the future," *IEEE Pervasive Comput.*, vol. 1, no. 2, pp. 76–82, Apr./Jun. 2002. doi: 10.1109/MPRV.2002.1012340.

[17] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proc. CASES*, Seoul, South Korea, 2006, pp. 234–241.

[18] A. C. Evans, K. Davis, J. Fogarty, and J. O. Wobbrock, "Group touch: Distinguishing tabletop users in group settings via statistical modeling of touch pairs," in *Proc. ACM CHI*, Denver, CO, USA, 2017, pp. 35–47.

[19] S. R. Richter, C. Holz, and P. Baudisch, "Bootstrapper: Recognizing tabletop users by their shoes," in *Proc. ACM CHI*, Austin, TX, USA, 2012, pp. 1249–1252.

[20] J. Park, I. Kim, and Y. I. Eom, "Grouping applications using geometrical information of applications on tabletop systems," in *Proc. ACM UIST Adjunct*, Quebec City, QC, Canada, 2017, pp. 181–182.

[21] J. Kay and P. Lauder, "A fair share scheduler," *Commun. ACM*, vol. 31, no. 1, pp. 44–55, Jan. 1988. doi: 10.1145/35043.35047.

Jonggyu Park received the B.S. degree in software and the M.S. degree in platform software from Sungkyunkwan University, South Korea, in 2014 and 2016, respectively, where he is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering. His research interests include storage systems and operating systems.

Young Ik Eom received the B.S., M.S., and Ph.D. degrees in computer science and statistics from Seoul National University, South Korea, in 1983, 1985, and 1991, respectively. He was a Visiting Scholar with the Department of Information and Computer Science, University of California at Irvine from 2000 to 2001. Since 1993, he has been a Professor with Sungkyunkwan University, South Korea. His research interests include virtualization, operating systems, storage systems, cloud systems, and UI/UX system.