# Resource Management for Web Applications in ServiceOS

Alexander Moshchuk and Helen J. Wang
{*alexmos, helenw*}*@microsoft.com*
*Microsoft Research, Redmond*

## Abstract

Recent work [42] has established the need to build a web browser as a multi-principal operating system where a principal is a web site. That work designed the protection architecture of such a browser. Another fundamental facility that an OS must offer is resource management, including both access control and resource sharing among authorized principals. Unfortunately, resource management in existing browsers is largely non-existent, and resource management of a commodity OS is ill-suited for many web applications that embed content from other principals.

In this paper, we tackle the problem of resource management for web applications in a multi-principal OS-based browser called *ServiceOS*. ServiceOS provides web applications with systematic and consistent access and control of devices, such as camera and GPS, and it uses a novel *DOM-recursive* resource allocation policy by default when resources like CPU and network bandwidth are under contention. We also introduce *application-specified* resource allocation to allow web programmers to explicitly influence resource management based on web application semantics.

We have built a ServiceOS prototype that manages a wide range of resources, including CPU, memory, network bandwidth, and devices like cameras, microphones, or GPS. Our evaluation shows that compared to existing browsers, ServiceOS provides web applications with improved service quality, fairness, and security.

## 1 Introduction

Web browsers have evolved into multi-principal operating environments where mutually distrusting web site principals share the underlying system resources [41]. To this end, Gazelle [42] has proposed to architect a web browser as a multi-principal operating system, supporting web site principals as first-class citizens. While Gazelle re-architected the protection architecture of a browser, we tackle another fundamental OS facility: resource management, including resource access, control, and sharing, for web applications in such an OS-based browser. Figure 1 shows the architecture of such a browser system; we provide more details in Section 2.

Resource management in existing browsers is largely non-existent. Today's browsers don't manage resources like CPU, memory, or network bandwidth for web applications. They also don't provide web applications with systematic and consistent access and control of common I/O devices, such as cameras or GPS. The lack of resource management directly impacts the capabilities and quality of today's web applications, making them less rich and less robust than their desktop counterparts. Recent work [32] suggests relying on a commodity OS's resource management for a multi-process-based browser. However, we argue that resource management mechanisms in commodity OSes can be ill-suited for many web applications.

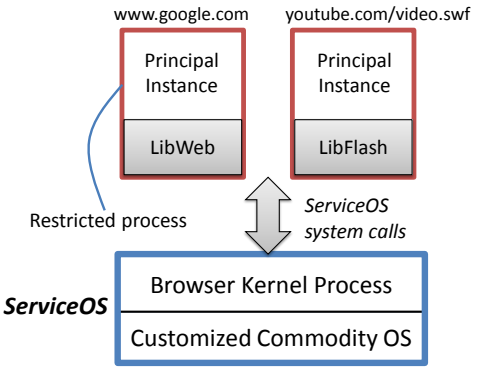


Figure 1: **Multi-principal OS-based browser architecture where a principal is a web site**: Principal instances' processes are restricted so that they can only issue ServiceOS system calls by IPC to the browser kernel process and cannot interact with the underlying commodity OS. The browser kernel process handles ServiceOS system calls and utilizes the commodity OS for system resource access and resource management as it pertains to web applications.

For resource sharing, one could consider mapping web site principals onto user principals of a commodity OS (i.e., putting different web site principals into separate processes and labeling each process with a unique uid). This is insufficient, as it is common for web applications to embed content from different principals, such as an advertisement, a user profile enclosed in an <iframe> tag, or a Flash movie enclosed in an <object> tag. Fundamentally, the fairness in resource management across principals in commodity OSes (e.g., in CPU scheduling) is not suited for web applications' recursive, cross-principal service composition because it would give a principal the ability to obtain arbitrary amounts of resources. As illustrated in Figure 2, a web application a.com embeds a principal attackerAd.com which

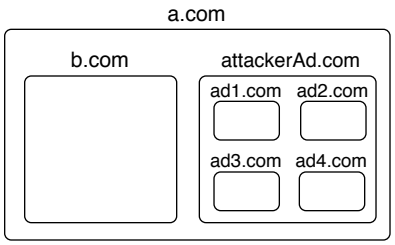


Figure 2: **Cross-principal service composition in web applications.** By embedding many services, attackerAd.com can mount a denial-of-service attack against a.com with current scheduling mechanisms in commodity OSes.

can in turn embed many different principals to cause denial-of-service to `a.com` with a commodity OS's CPU scheduling mechanisms.

For resource access control, web applications require prudent considerations: for example, an untrusted site should never access a privacy-sensitive device, such as a microphone. Access control for web browsers also requires easy extensibility to new devices and new web APIs for accessing them, scalability to a large and growing number of web principals, and simplicity in management. Access control mechanisms in commodity OSes were never designed to achieve these goals. Today, browser plug-ins, such as Flash or Google Gears, are allowed to directly interact with the underlying OS to access devices. Access control is then at the discretion of each plug-in. This is flawed, because it results in many security policies coexisting, and often conflicting, within the same browser. Worse, the compromise of *any* plug-in allows a malicious web site to bypass the plug-in security model and access any devices at will.

In this paper, we devise resource management for web applications in the context of *ServiceOS*[1], a multi-principal OS that supports web applications as first-class principals. For access control, we separate access control policies from resource access mechanisms and centralize all access control decisions for all devices. Our key idea for managing access is a Resource Object Model (ROM) API for access control that is aware of device semantics. The ROM eases extensibility to new devices and simplifies access control management with flexible access control granularities.

For resource sharing, when under resource contention, we use *DOM-recursive* resource allocation policies *by default* for divisible resources like CPU and network bandwidth. The recursion hierarchy corresponds to the principal embedding hierarchy of a web application, allowing *fair* sharing of resources in the face of cross-principal web application composition. The principal embedding hierarchy typically corresponds to the HTML DOM [8] tree's hierarchical relationship between the involved principals, hence the name "DOM-recursive". In addition, we introduce *application-specified* resource allocation so that a programmer can maximize a web application's performance by explicitly allocating resources to embedded principals. Together, the DOM-recursive and application-specified policies provide the optimal balance between fairness and efficiency.

We have built a Windows-based ServiceOS prototype that manages a wide range of resources, including CPU, memory, storage, network bandwidth, and peripheral devices like cameras, microphones, or GPS. Our evaluation shows that compared to existing browsers, ServiceOS provides web applications with improved service quality, fairness, and security.

Section 2 gives the background on multi-principal OS-based browser architecture. We establish our conceptual framework in Section 3, categorizing resources according to their semantics and deriving the appropriate access control and resource sharing approach for each category. We present our design on resource sharing in Section 4 and on access control in Section 5. We describe our prototype implementation in Section 6 and present our evaluation in Section 7. We compare and contrast with related work on web browsers and resource management in Section 8. Finally, we conclude in Section 9.

## 2 Multi-Principal OS-Based Browser Architecture

In this section, we describe the architecture of ServiceOS, which is a multi-principal OS that supports web applications as first-class principals. Our architecture, shown in Figure 1, is based on the Gazelle browser's architecture [42] with slight adaptations for the purposes of resource management.

Unlike traditional OSes, which treat users as principals, ServiceOS treats *web applications* or web sites as OS principals, as in Gazelle [42] and MashupOS [41]. The principal is labeled with the triple of <`protocol`, `domain`, `port`>, just as in the same-origin policy [35] in today's browsers (but today's browsers are *not* constructed as multi-principal OSes [42]). A web site principal can embed other principals through frame and object tags.

A principal is the *unit of protection*. Principals are completely isolated in resource access and usage. Any sharing must be made explicit. Just as in desktop applications, where instances of an application are run in separate processes for failure containment and independent resource allocation, a principal instance is the *unit of failure containment* and the *unit of resource allocation*. For example, navigating to the same URL in different

[1]The "service" in ServiceOS comes from the "service" in "Software-as-a-Service".

tabs corresponds to two instances of the same principal. When `a.com` embeds two `b.com` iframes, we assume the two iframes are independent and run them using two instances of `b.com` [2]. However, if `a.com` embeds a same-origin frame, that frame runs in the same `a.com` principal instance as the host page by default; we allow the host page to change the default and designate an embedded same-origin frame or object as a separate principal instance for independent resource allocation and failure containment. Principal instances are isolated for all runtime resources, but principal instances of the same principal share persistent state such as cookies and other local storage. Protection unit, resource allocation unit, and failure containment unit can each use a different mechanism depending on the system implementation. Because the implementation of our principal instances contains native code, we use OS processes for all three purposes.

ServiceOS consists of both a browser kernel (BK) process and a possibly customized commodity OS. The BK process handles the ServiceOS system calls from principal instances and utilizes the commodity OS to access system resources. The logic for processing web content (i.e., libweb, plug-ins) resides in the principal's address space, and the BK is agnostic to web content semantics. Resource management functionality is mostly realized in the BK, which relies on underlying OS mechanisms to impose its resource sharing policies. If the OS does not provide the necessary resource sharing mechanisms, it may need to be customized. For example, for our Windows-based prototype, we needed to customize the CPU scheduling in Windows, but had we chosen to build a Linux-based prototype, we could likely leverage Linux's existing group scheduling facility [5].

The processes of principal instances are restricted so that they cannot interact with the underlying OS. To access resources, they must use ServiceOS system calls. Because plug-ins run in a web principal's address space, they are restricted to the same ServiceOS system calls. The feasibility of such process restriction has been demonstrated by the picoprocess design and implementation in Xax [9].

## 3 Conceptual Framework

Resource management consists of access control and resource sharing. The access control for a resource determines which principals can access the resource. Resource sharing manages resource contention, namely, how the resource should be shared among contending authorized principals or their principal instances. Access control and resource sharing are orthogonal to each other and can be designed independently. Next, we map out the resource characteristics for both access control and resource sharing and pinpoint the appropriate approaches.

**Access Control.** For access control, we have categorized resources into *basic computing resources*, *user input devices*, and *peripheral resources*, as shown in Table 1. Access control on basic computing resources, such as CPU, memory, display, network bandwidth, and storage, is a matter of principal admittance to the system. The browser kernel can possibly adopt a whitelist or blacklist-based filter (e.g., for anti-phishing purposes), denying some principals from being admitted to the system. Once a principal is admitted, it can access all of the basic computing resources.

User input devices like mouse, keyboard, or touch screen can only be used by the user, and cannot be used by any web site principals; otherwise web site principals could impersonate the user.

Peripheral resources, such as those listed in the table, are subject to discretionary access control for which we present our design in Section 5.

**Resource Sharing.** For resource sharing, we first divide resources into *runtime* resources and *persistent* resources, as shown in Table 2. Runtime resources are only available when the browser application is running, while persistent resources like storage survive browser restarts.

A browser's storage resources include cookies and more recently client-side local storage and application cache as proposed by HTML5 [18]. We observe that only the user can be responsible for reclaiming and arbitrating the storage space among principals when storage is under contention (e.g., running out of disk space). For example, a user may prefer all of her address book to be present at all time, even if the address book web application has exceeded a fair share of the storage space.

We further subdivide runtime resources into *divisible* and *non-divisible* resources. A divisible resource allows fractions of the resource (whether in time dimension or space dimension) to be shared or allocated among the contending principal instances. For divisible resources, we apply DOM-recursive resource allocation and application-specified resource allocation when under resource contention; we present both in detail in Section 4.

ServiceOS considers input sensors like GPS, accelerometer, or compass as *transactional*, because such a device's interactions are independent from one another and the device's data can be obtained with a single system call; each use is a short transaction (rather than a long session that needs to be explicitly set up and torn down). Such resources require no additional sharing

[2] Per HTML5 [18] specification, such related same-origin frames can access each other's DOM and must share the same event loop. By default, our model schedules them independently using different renderers and relies on IPC for communication, but to support existing semantics, we allow a `b.com` frame to specify that a related `b.com` frame should share its principal instance.

| Resource types | Resource Examples | Access Control Mechanism |
|---|---|---|
| basic computing resource | CPU, memory, display, data network, storage | principal admittance in BK |
| user input devices | mouse, keyboard, touch screen | exclusive user access |
| peripheral resources | camera, GPS, telecom network, microphone, speaker | discretionary access control (Section 5) |

Table 1: **Resource characterization for access control.**

| Resource types | | Resource Examples | Resource Sharing Mechanism |
|---|---|---|---|
| runtime resources | divisible | CPU, memory, display, data network, input sensors (GPS, accelerometer, compass) | DOM-recursive sharing and application-specified resource allocation (Section 4) |
| | non-divisible | session-based resources (telecom network, microphone, speaker, camera) | user-arbitrated sharing |
| persistent resources | | storage resources (disk, flash) | user-arbitrated sharing |

Table 2: **Resource characterization for resource sharing.**

policies since they are effectively scheduled by the policies controlling CPU and memory — the BK accesses them by executing the sensor's read system call code (abiding by CPU scheduling rules for principal instance that is making this request) and reading buffered sensor data from memory.

Non-divisible resources cannot be divided in time or space. Session-based resources in the second row of Table 2 are examples of such resources. When such resources are under contention, users need to decide the winner based on their needs. For example, the user is on the phone using the telecom network through one application; if another application needs to make a phone call, the browser kernel must prompt the user to make a decision to either stay on the current call or make a new call. Similarly, if a web application is in an exclusive video camera recording session, only the user can be the decision to interrupt it and grant camera access to another application.

An allocated fraction of certain resources is accessible to only one authorized principal instance or principal at a time. For example, a CPU time slice or a piece of memory allocated for a principal instance can only be used by that principal instance. On the other hand, depending on device support, resources like speaker, microphone, and video camera could offer simultaneous access from multiple principal instances. For example, several principal instances may be fine with outputting to the speaker concurrently and having their audio data mixed together; audio and video streams from microphone and video camera may be received simultaneously by multiple principal instances for various recording purposes. Table 3 summarizes exclusive-use and concurrent-use resources. For concurrent-use resources, we allow web applications to choose between concurrent and exclusive use. Our system provides defaults for all such devices, and web applications are allowed to change the default later. For example, as indicated in Section 5's Table 4, our system permits concurrent use for speaker by default, but not microphone for privacy reasons. Note that because of their transactional read-only nature, input sensors like GPS can always be read concurrently.

| | |
|---|---|
| Exclusive use for principal instance | CPU, memory, display, network bandwidth, dialer |
| Exclusive use for principal | storage |
| Concurrent use | speaker, microphone, camera, GPS, accelerometer, compass |

Table 3: **Exclusive use vs. concurrent use for an allocated fraction of a resource.**

## 4 Resource Sharing

In this section, we present our resource sharing design for divisible resources, including CPU, memory, display, and data network bandwidth.

Resource sharing in ServiceOS is performed on principal instances, which form the unit of our resource allocation (Section 2). For example, in Figure 3, `a.com` embeds content from two other principals, `b.com` and `c.com`, and `c.com` further embeds content from `d.com`. This corresponds to four principal instances with one instance for each site.

### 4.1 DOM-recursive resource allocation

In the face of resource contention, the straightforward "flat" scheduling across contending principal instances (as in round-robin CPU scheduling for desktop applications in Windows or Linux) is not well-suited for Web applications. Unlike desktop applications, a Web application can embed content from other principals; flat scheduling would enable a Web application to acquire an arbitrary amount of resources, thereby conducting a

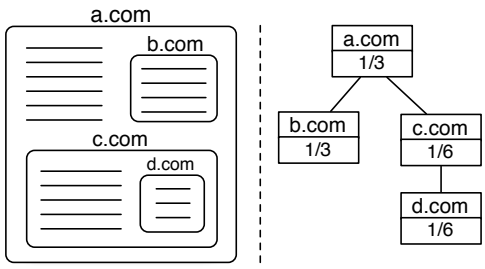


Figure 3: **DOM-recursive sharing in the presence of web application composition.** By default, ServiceOS allocates resources according to the hierarchical share of each contending principal instance.

denial-of-service attack. Figure 2 gives such an example.

Instead, we must consider the hierarchical organization of content on today's Web pages. To this end, ServiceOS uses *DOM-recursive* allocation, where the resource is divided recursively along the principal embedding hierarchy of a web application. Namely, ServiceOS first divides the resource into equal fractions among the top-level contending principal instances, each of which runs in a separate browser tab. The fraction of the resource assigned to a principal instance is then split fairly among the principal instance itself and all of its embedded principal instances that are actively contending for the resource. We give the pseudo code for the recursive `allocate` function below, which calculates the resource share for each contending principal instance.

```
allocate(null, top_level_principal_instances, 1);

function allocate(parent, children, amount) {
    numChildren = |children|;
    if (parent != null) {
      numChildren++;
      share[parent] = amount/numChildren;
    }
    foreach principalInstance in (children) {
      share[principalInstance] = amount/numChildren;
      if (hasChildren(principalInstance))
          allocate (principalInstance,
                    principalInstance.children,
                    share[principalInstance]);
    }
}
```

As an example, in Figure 3, assuming that `a.com` is the only top-level web page (i.e., only one tab is opened), if all principal instances are actively contending the resource, `a.com`, `b.com`, and `c.com` will each receive 1/3 of a resource; and `c.com` splits its 1/3 among itself and `d.com`, so that each will get 1/6 of the resource. If `b.com` is not contending the resource, then `a.com` and `c.com` both receive 1/2, where `c.com` splits its share for itself (1/4) and `d.com` (1/4).

The key property of our DOM-recursive resource allocation policy is that it provides fairness across all the contending principal instances at the *same level* of the principal embedding hierarchy, but not across *all* the principal instances. We use this policy by default for allocating CPU and network bandwidth (but *not* memory, as we will see shortly).

We separate resource allocation policy from resource allocation mechanisms. We use the same DOM-recursive resource allocation policy for all the relevant resources, but resource allocation mechanism is different from each resource. For CPU scheduling, existing proportional scheduling mechanisms, such as a lottery scheduler [39] or a hierarchical scheduler [5, 28], can be used and fed with our policy; we used the former in our implementation. For the data network bandwidth allocation mechanism, various bandwidth shapers [34, 10, 27] can be used to fairly share bandwidth across contending principal instances; we used NetLimiter [27]. Bandwidth contention can happen when the last-hop bandwidth is limited. For example, 3G networks offer an average of only 841 Kbps [45].

In Section 7, we give realistic examples of how existing browsers, including Gazelle and Google Chrome, allow resources to be taken over by misbehaving principals embedded in a web application, and how ServiceOS can handle such situations gracefully.

**Memory.** Complex Web content may require significant memory resources for processing. Unfortunately, a script on a Web page (or in an embedded gadget) could exhaust the host's available memory, causing thrashing or denying memory allocation to scripts in concurrently executing Web content. The browser must manage memory to prevent such scenarios; for example, it seems plausible for the browser to terminate an embedded untrusted ad that is stealing memory from the parent page and other top-level pages. However, this turns out to be a hard problem. Memory is difficult to reclaim once allocated, yet denying a memory allocation ultimately denies execution. Most browsers, such as Internet Explorer, choose to ignore such problems, relying on the user to close memory-intensive tabs; Google Chrome additionally enforces an arbitrary upper memory limit for any top-level Web page, terminating all of its scripts once the limit is exceeded. No browsers attempt to fairly manage memory among gadgets embedded on the same Web page.

In trying to improve upon these approaches, we considered and rejected several approaches. First, we considered using DOM-recursive allocation for custom working set management when memory is under contention [16, 24, 22]. We rejected this approach because paging renders an application's performance intolerable, with little room for improvement. We also tried to have the browser kernel permissively allocate mem-

ory to all principal instances; when out of memory, the BK could simply terminate lowest-priority principal instances (e.g., embedded ads) to reclaim memory. Unfortunately, determining the relative importance of various embedded content is intractable for an arbitrary Web page since the BK cannot infer the page's semantics. Therefore, terminating an embedded gadget might break the functionality of another component on the parent page.

Overall, we found that to improve memory management within top-level tabs, the browser must know more about a Web page's resource usage semantics. Section 4.2 describes how Gazelle allows pages to express such semantics, including new abstractions for specifying memory requirements of various elements of a page.

**Display.** Fairly sharing the display is meaningless for web applications. In fact, display is typically allocated by a parent principal instance to its children through explicit position and dimension specifications, such as width and height attributes of frames and object tags. Gazelle [42] gives a comprehensive design of display management and protection; ServiceOS adopts Gazelle's display protection design.

**Dynamically-generated content.** Today's Web pages can execute code (e.g., JavaScript's `document.createElement`) to dynamically inject new embedded content and new principal instances. Gazelle properly readjusts its policies when such events occur. Dynamically spawned principal instances result in new processes created by the browser kernel, and the BK recalculates the hierarchical resource shares used for CPU and network allocation.

## 4.2 Application-Specified Allocation

In the absence of knowing web applications' resource needs, DOM-recursive resource allocation policy is fair and prevents an embedded, misbehaving principal from monopolizing resources. Nevertheless, a web application itself may know how to optimally allocate resources among itself and embedded content. For example, if `a.com` embeds an untrusted ad, it might want to restrict the ad's CPU and memory usage; if `a.com` embeds a video application, it might indicate that this video can be best viewed with a bandwidth of 200 Kbps. To support such scenarios, we enable web programmers to provide *application-specified* resource allocation for their web applications. Note that this does not violate hierarchical fairness: on Figure 3, `c.com`'s resource policies can influence `c.com` and `d.com`, but they cannot affect resources granted to `a.com` or `b.com`.

For CPU and network bandwidth, we allow an application to specify a resource share for each of its embedded applications; this allows the application to grant relative priorities among its children and itself. The fractions "zero" and "1" are interpreted as an absolute priority.

For memory, we allow an application to customize the priority level of an embedded application to "disposable". This indicates that the embedded application can be safely terminated to reclaim memory if memory contention arises. For example, a Web application may embed an ad as follows: `<iframe src='http://ad.com' cpu=0 memory=disposable>`. This indicates that the ad shouldn't run when in CPU contention, and it should be terminated when under memory contention.

We also allow preferred network bandwidth to be expressed with a `preferredBW` attribute, which indicates the bandwidth needed for optimal service quality. An embedded application with this specification will receive higher priority in receiving its preferred bandwidth amount or the available bandwidth, whichever is smaller. When multiple embedded applications are given preferred bandwidth specifications and when the system cannot satisfy all their needs, the bandwidth allocation will follow their relative proportions.

Currently, we assume that the creator of the includer page knows the proper `preferredBW`. More elaborate schemes, such as a negotiation process between the hosting page and the site providing the video, could be explored in future work.

Custom allocations can also be adjusted programmatically at the runtime; only the principal who sets the allocation can change the allocation. These custom allocations result in system calls to the browser kernel; the BK then sets these policies for all resource allocation mechanisms.

One last consideration is mission-critical applications like those that use a voice network (e.g., to receive a phone call). These types of applications are granted absolute priority in all resource use on ServiceOS.

# 5 Discretionary Access Control

In this section, we present our access control design for peripheral resources (Table 1).

## 5.1 Systematic Access Control

Unlike ad-hoc resource access control in today's commodity OSes (see Section 8), we want all web applications' accesses to *any* peripheral resource to adhere to a centrally-enforced access control policy. We achieve such systematic access control by separating access control policies from resource access mechanisms. As illustrated in Figure 4, we build the access control layer as a reference monitor [2], where the BK decides whether

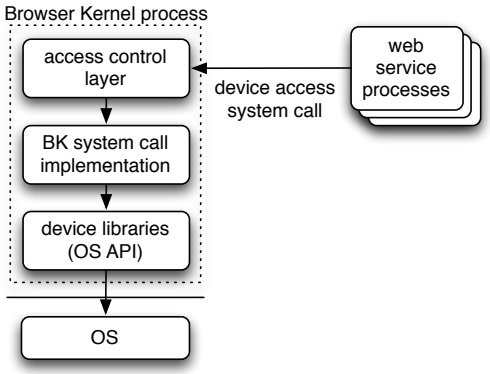


Figure 4: **The flow of a resource access system call.** The browser kernel separates access control from the implementation of resource-accessing system calls.

*each* system call is permitted before ever reaching the call's implementation.

In contrast, some OSes place access control checks inside the kernel's system call implementation. Our design has the advantages of 1) *single point of access control*: since access control is separated into its own module, access control policies and their implementations are less error-prone and easier to validate, and 2) *extensibility*: support for new devices can be added to the BK without touching the access control layer or understanding the notion of web principals.

## 5.2 Resource Object Model

The rapidly evolving browsers and development of new Web standards like HTML5 necessitate an access control system that is both *extensible* and *manageable*. For extensibility, ServiceOS must be able to gracefully adapt to newly-added devices or device APIs; for example, the number of changes in access control state should be minimized. For manageability, ServiceOS must strike a balance in access control granularity. While fine granularity is precise at granting only what is needed by a web application, coarse granularity simplifies manageability by reducing the number of access control operations.

To support these goals, we followed the principle of *separating resource semantics (e.g., location) from physical resources (e.g., GPS)*. More specifically, we organize the resource functionality hierarchically by semantics into a *resource object model* (ROM) tree structure (similar to DOM). Figure 5 gives an example ROM tree. Leaves represent the most fine-grained functionality. Related resource functions at the same tree level are grouped together into higher-level nodes in the tree, forming more coarse-grained resource functionality that can be granted. The root node groups all resources in the system. Access to any node of the tree can be granted to a web application; the leaf nodes in the subtree rooted by

| ROM Nodes | attributes | | | |
|---|---|---|---|---|
| | userask | background | lifetime | concurrent use |
| Speaker | don't ask | allow | forever | yes |
| Location | ask | allow | session | n/a |
| Photo camera | ask | deny | session | n/a |
| Microphone | ask | allow | session | no |
| Dialer | ask | deny | one-time | n/a |

Table 4: **A default list of access rights handled out to a web application.**

that node represent all the functions granted to the web application.

The separation of resource semantics from physical resources eases extensibility when adding a new physical resource that follows existing resource semantics. For example, upgrading a GPS device to a newer model with a new driver will not affect the ROM tree and its access control at all; similarly, adding a new type of device that provides location information under the "location" node will still obey access control rules for the "location" node.

The specific mechanism for implementing access control for ROM nodes could rely on ACLs, capabilities [23, 6, 37], or both. We detail our current implementation in Section 6.5.

An access rule on a ROM node uses attributes that indicate *how* access can be performed on a resource. Currently, we support four such attributes: *userask*, *background*, *lifetime*, and *concurrent use*. The boolean *userask* attribute instructs the BK to either ask or not ask the user to confirm the resource access. The *background* attribute instructs the BK to deny or allow resource access when a web application runs in tabs which are minimized, not focused, or otherwise not visible to the user. The *lifetime* attribute can take the values "one-time", "session", "forever" and informs the BK that access to a ROM node is for either (1) one-time use (the rule will be deleted after first use), (2) the duration of the current browsing session of a particular principal instance (the rule will be deleted after the user navigates away or closes the window containing the principal instance), or (3) forever (the rule will survive browser restarts and apply to all future instances of this principal). The *concurrent use* attribute is for resources that allow concurrent access from multiple principal instances (see end of Section 3): when its value is "yes", it allows other principal instances to use the resource concurrently with this principal instance; when "no", it denies access from other principal instances when this principal's instances are using the resource. The attribute list can be easily extended to support other kinds of generic restrictions, such as an ability to charge money (e.g., when dialing toll numbers on a telecom network).

When a principal instance begins executing, the BK grants it a list of default access rights, a sample list of

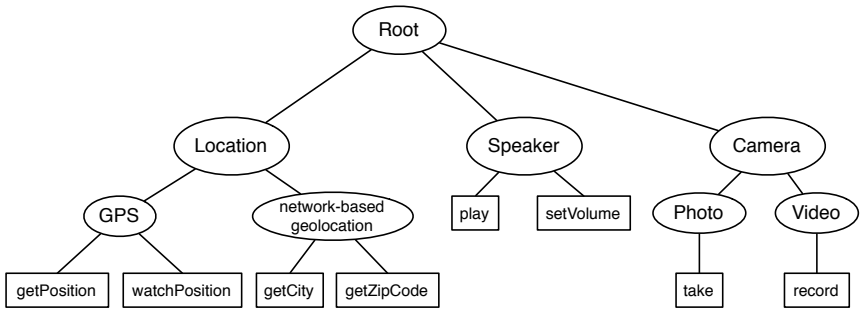


Figure 5: **A partial Resource Object Model (ROM) tree.** The ROM organizes all resource functionality into a semantically meaningful hierarchy.

such rights is shown in Table 4. A web application might want to obtain additional access rights for a resource. For example, it might want to avoid asking the user every time it takes a picture (replace an *ask* Camera access right with a *don't ask* Camera access right), or it might want to obtain permission to use a GPS in the background. ServiceOS allows web applications to expand their access rights for a resource with user approval. We adopt *manifests* [21, 19, 12] as a mechanism for centralizing all decisions involving the user and reducing the number of user prompts to at most one. Any web application can provide a manifest specifying required resource functionality; this is presented to the user when the application loads and before any device access is performed. The user can change the *userask* and other attributes for any access right the service already possesses [3]. However, the user does *not* have the ability to issue arbitrary new access rights to a web application by default. This can protect access to very sensitive resources where users might not be trusted with an access decision.

# 6 Implementation

We have implemented the ServiceOS architecture as shown in Figure 1. We implemented the browser architecture on Windows Vista, adopting Gazelle's design [42]. We only describe customizations to Windows and additions to the browser kernel which enabled our resource management design. CPU scheduling required customizing the Windows kernel, which lacks hierarchical scheduling abstractions. To ease this effort, we implemented and evaluated CPU scheduling on the Windows CE 6.0 kernel, which is supported by Gazelle's browser kernel in addition to Windows Vista. We implemented resource sharing for bandwidth and transactional devices, as well as access control, in the browser kernel. Our browser kernel can run on either Windows CE or Windows Vista, but due to current implementation limitations, device access by principal instances is only supported on Windows Vista. Therefore, our CPU scheduling experiments are conducted on Windows CE, while other experiments use full-featured principal instances on Windows Vista (Section 7).

[3]The way in which the user is asked for these security decisions is critically important but out-of-scope for this paper; novel ideas are explored in recent work [17].

## 6.1 CPU Scheduling

We modified the Windows CE kernel to implement lottery scheduling [39]. By default, Windows CE uses round-robin scheduling among 256 priority queues. Many system-critical tasks, such as those servicing I/O interrupts and manipulating hardware devices, run at high kernel-reserved priorities; to avoid disrupting these services, we implement our lottery scheduler at a dedicated priority level which is equivalent to applications' default level. More details on a similar implementation are described by Petrou et al [29].

The browser kernel is treated as a special process by the scheduler: it is given absolute priority over all the processes of principal instances, it instructs the CE kernel to enter new principal instances into the lottery scheduler, and only its process is allowed to adjust ticket values for principal instances. The scheduler first holds a lottery to pick a principal instance process, and then it runs the process's threads (e.g., threads for JavaScript engine, rendering, or IPC) round-robin. The browser kernel enforces application-specified resource allocation policies by adjusting ticket amounts. We have not yet implemented support for application-specified absolute priority.

One limitation of our current implementation is that any CPU used by the browser kernel process on behalf of a principal instance process (e.g., when processing a system call) is not charged to the principal instance. We could easily address this by modifying the BK to provide a separate request-handling thread for each principal instance, and have that thread share the principal instance's lottery tickets; many ways to solve this problem have been studied [11, 3, 33, 15].

## 6.2 Memory

We implemented support for memory priorities in the browser kernel using the Process Status API exposed by Windows to obtain physical memory usage of a process. The browser kernel inspects memory usage of all principal instances and invokes application-specified termination algorithms when physical memory runs short. We determine memory shortage conservatively before the OS kernel engages its own algorithms when physical memory runs out. When calculating a principal instance's memory usage, the BK only considers pages which are not shared with other instances.

## 6.3 Bandwidth

We used the NetLimiter [27] tool, which is a kernel driver, to implement our bandwidth sharing policy. The browser kernel programmatically manipulates NetLimiter's *grant* and *limit* parameters for principal instances' network connections to hierarchically enforce bandwidth limits when under contention and to enforce application-specified bandwidth allocation policies.

## 6.4 Device access

We have implemented the Resource Object Model APIs (Section 5) for accessing cameras, GPS, local storage, and network connectivity detection. We have exposed access to these resources to Web applications through the JavaScript DOM, defining device access APIs similar to recommendations by W3C [30] and OpenAjax [25]. Our architecture also requires us to port plug-ins like Flash to use our device access system calls. We leave this as future work; our evaluation in Section 7.3 demonstrates that our architecture would impose a very small performance hit for plug-ins compared to their native device access.

## 6.5 Access control mechanisms

For imposing access control policies, we considered using ACLs and capabilities. Each mechanism has its advantages and drawbacks. ACLs require maintaining a list of authorized principals for each resource, raising complexity in managing a very large and constantly growing number of web site principals. Capabilities present another set of challenges, such as revocation, determining which sites have access to a resource, and the implicit design principle that a capability can be passed around, which would allow one web application to grant undesirable device access to arbitrary other web applications.

In ServiceOS, we have currently adopted a hybrid approach which is based on ACLs but borrows some ideas from capabilities. All access control rules are located in and maintained by the browser kernel exclusively. When a new principal instance starts executing, the browser kernel establishes a set of its access rights by combining a default-access list (Table 4), which is defined by the browser vendor and the user, with any of the principal's persistent capabilities (those with a "forever" attribute). The former specifies default rules for each ROM node (like ACLs), the latter is attached to each web principal (like capabilities) and takes priority in determining any access. Although the number of per-principal lists of rules can become large, we only expect a fraction of web sites to use device access and still fewer to be granted persistent access to a device.

# 7 Evaluation

In this section, we evaluate our prototype and demonstrate the following key points:

- Unlike existing browsers, ServiceOS can effectively insulate CPU usage of a web page from untrusted content it embeds.
- Unlike existing browsers, ServiceOS can fairly allocate network bandwidth among two network-intensive gadgets (i.e., applications) embedded on the same web page. Furthermore, a custom policy allows a gadget to preferentially receive more bandwidth; this improved one streaming-video gadget's throughput by 4.6x over a default Windows policy and eliminated the buffering encountered during video playback.
- Our prototype performs well for a sophisticated, real-world AJAX web application which accesses several devices, such as a camera. We find that device access in ServiceOS is only 5% slower than that of a native application or a plug-in.

We run these experiments on our ServiceOS prototype on a Windows Vista machine with a 3GHz dual-core processor, 4GB RAM, and a gigabit Internet connection. For our CPU experiments, we used a Windows CE device emulator running on the above system.

## 7.1 CPU scheduling

We constructed a computation-intensive microbenchmark application `app.com` which is in CPU contention with a misbehaving `ad.com` that it embeds. The ad contains an infinite loop of expensive image swapping operations, and it embeds three other ads which perform the same computation. The resulting web application composition is shown in Figure 6. We measure performance by adapting the *regexp* JavaScript benchmark distributed with the V8 JavaScript Engine [38]: we define

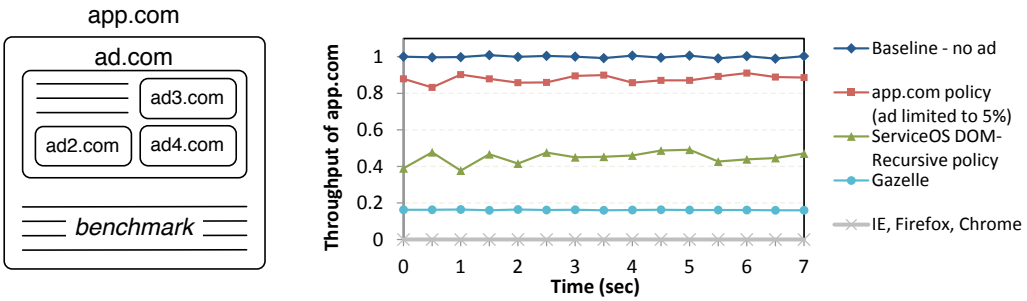


Figure 6: **Throughput of `app.com` when in CPU contention with a malicious principal `ad.com`.** ServiceOS allows `app.com` to regain most of the throughput it loses to `ad.com` in today's browsers.

`app.com`'s *throughput* as the number of benchmark iterations it is able to complete per second. We normalize throughput measurements according to the baseline observed when `app.com` executes without `ad.com`.

All popular browsers, including Chrome, Firefox, and Internet Explorer, place all five web principals in the above scenario into the *same* process and execute them using the same instance of the JavaScript engine (despite the fact that they are from different principals and can run concurrently [32]). If `ad.com`'s computation runs before `app.com` starts its benchmark (as in many sites with top-placed banner ads), `app.com`'s code never runs and has a throughput of 0. In contrast, both Gazelle and ServiceOS run `app.com` and `ad.com` in separate processes using different JavaScript runtimes. However, this in itself is *insufficient* to guarantee CPU fairness: Figure 6 shows that in Gazelle, `app.com` achieves a throughput of only 0.17 on average, because the OS splits CPU time fairly between `app.com` and four malicious ad principals.

In contrast, under ServiceOS's default DOM-recursive sharing policy, `app.com`'s throughput improves to 0.45, which gives the fairness that is absent in the existing browsers. This is because DOM-recursive sharing gives 50% of the CPU to `app.com` and splits the other 50% among ad.com and its children. Furthermore, if `app.com` takes advantage of our application-specified QoS mechanisms and limits the untrusted `ad.com` iframe to 5% of the CPU, its throughput jumps to 0.88. This demonstrates that ServiceOS can accurately abide by application-specified CPU allocation.

The slight variations in throughput over time observed for our ServiceOS prototype are caused by the probabilistic nature of our lottery-based scheduling implementation. Note that all scheduling strategies involving several processes incur Windows CE's overhead of context switching. This overhead is responsible for making `app.com`'s throughput to be somewhat lower (e.g., 0.44 instead of 0.50) than what our scheduling policy targets.

We measured the overhead of using our implementation of lottery scheduling to be negligible. In our measurements, we adjusted amounts of tickets to simulate the underlying OS's round-robin scheduling. We found


| | gadget.com | video.com |
|---|---|---|
| Existing browsers | 687 Kbps | 172 Kbps |
| ServiceOS DOM-recursive policy | 440 Kbps | 404 Kbps |
| Custom app.com policy (video.com:800Kbps) | 45 Kbps | 802 Kbps |

Table 5: **Network throughput of `gadget.com` and `video.com`, both embedded by `app.com`.** Unlike Windows's default policy, ServiceOS divides network bandwidth fairly across `gadget.com` and `video.com` when they are bandwidth contention; `video.com` can further improve video playback with a custom policy specified by `app.com`.

that the throughput of `app.com` under a lottery-backed scheduling is only 0.9% less than the throughput under Windows's default scheduling.

## 7.2 Network bandwidth

Today's browsers rely on the underlying OS to manage network bandwidth contention among TCP connections made by web applications. To demonstrate why this approach is insufficient, we created another simple web application, `app.com`, which embeds two bandwidth-intensive gadgets: `gadget.com` and `video.com`. `gadget.com` continually downloads and processes four concurrent TCP streams of XML data using XmlHttpRequest, and `video.com` embeds a 52-second streaming Flash video file from `hulu.com`. To observe the network bandwidth under contention, we use NetLimiter [27] to limit the host's download capacity to 841 Kbps (average bandwidth available to smartphones on a 3G network [45]).

Our results are shown in Table 5. We first measured the average network throughput over three minutes for both `gadget.com` and `video.com` using Internet Explorer on our Windows system. We found that as expected, all TCP connections share available bandwidth fairly equally, and since `gadget.com` is using four connections, it receives 4x more bandwidth than `video.com`, leaving the latter with only 20% of available bandwidth. This significantly degrades performance

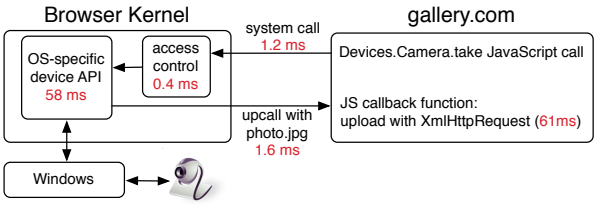


Figure 7: **Latencies involved when gallery.com takes a photo.** ServiceOS introduces only a 3.4 ms overhead (5%) on top of OS-specific device processing code.

of `video.com`, causing video playback to become unusable due to frequent stops for buffering. Overall, the 52-second video stream stopped for an additional 128 seconds of buffering. We verified the same behavior occurs with other browsers, including Firefox, Google Chrome, and unmodified Gazelle.

In contrast, ServiceOS's default policy allocates bandwidth fairly among `app.com`, `gadget.com`, and `video.com`. Since only the latter two are in bandwidth contention here, they each get roughly half of the bandwidth, and `gadget.com`'s four connections split only `gadget.com`'s half of bandwidth, as Table 5 confirms.

The additional throughput is enough to improve, but not resolve choppiness of video playback, reducing the amount of buffering to 33 seconds. We next modify `app.com` to specify that the `video.com` <iframe> requires 800 Kbps of bandwidth, which is the target bitrate of the video we tested, and check how well ServiceOS adapts to this custom policy. The last row of Table 5 shows that `video.com` now receives 802 Kbps, matching the custom bandwidth specification. We indeed observed that the video stream can now play back without any buffering.

## 7.3 Device access

In this section, we evaluate performance of web applications that use ServiceOS's device access mechanisms. We focus on these main questions: 1) How much overhead does ServiceOS impose on device access, compared to a native desktop application or a plug-in? 2) Which components are responsible for that overhead? 3) Is our architecture able to handle throughput-intensive or latency-sensitive device access?

To answer these questions, we wrote a "photo gallery" web application, which organizes a user's photos. The application can use an attached camera to take photos directly from the web page and automatically upload them into its cloud database[4]. To take a photo, the page calls a JavaScript function `navigator.devices.camera.take`, which translates into a system call to the browser kernel. Upon receiving the system call, the BK verifies access control, accesses the camera using Windows video capture APIs (analogous to drivers for the BK), converts the resulting data to a JPEG image, and returns the resulting bits to the web page via an asynchronous upcall, which invokes a JavaScript callback function. The web page then uploads the photo into the cloud (using `XmlHttpRequest`) and adds it to the photo layout on the web page.

**Comparison with a native application.** To test our gallery application, we have attached a Logitech Webcam, which produces 35KB 640x480 JPEG photos, to our test system, and found that 61.2 ms passes between the time a user clicks on the "take" button and the time the web page receives the image data. We compared this latency to a native application, which used the same APIs to obtain the photo – that application's latency was 58ms. Thus, ServiceOS only imposed a 3.1 ms overhead (5%) for accessing the camera.

Figure 7 breaks down the source of this small overhead across various ServiceOS modules. We see a small 0.4 ms latency for access control, 1.2 ms for the system call request, and 1.6 ms for the system call response that includes the image data. The latency for code that converts JavaScript device functions into BK system calls is negligible. For our camera, these latencies are dominated by the actual device access time (58 ms) and the time spend uploading the photo (61 ms).

**Performance of BK system calls.** The above suggests that the IPC used by our BK system calls might be a potential source of overhead for larger data transfers in other devices. To see how our system call implementation scales for large transfers, we measured the throughput of continually accessing 8MB chunks of data from a web application's local storage. We found that our system call implementation can transfer data from BK to a web application at a rate of 43.2 MB/sec. This is sufficient to handle many throughput-intensive devices. For example, the maximum read throughput for modern Flash drives under ideal conditions is about 30 MB/sec [1], and a typical 640x480 video camera requires a throughput of 8.75 MB/sec without any compression [43].

To verify the absence of a throughput bottleneck in a real-world scenario, we have modified our photo gallery to provide video streaming functionality by continuously taking and uploading photos into the cloud, and we wrote a web page, which uses a simple AJAX-based protocol [26] to download and render this photo stream on a second client. Even with this simple streaming method, the second client achieved an impressive rate of 12 frames per second for the 640x480 stream, limited only by how fast photos could be obtained and uploaded. Enhancing ServiceOS's device APIs with a video capture

[4]We imagine such an application would be convenient on camera smartphones.

abstraction that provides data using a video-optimized encoder is certain to improve the frame rate for such an application.

Although we have not yet ported plug-ins to run on ServiceOS, these results provide evidence that plug-ins should not suffer major performance setbacks when accessing devices through Gazelle's new system calls.

**Ease of development.** For device access, web applications need to use ServiceOS's device APIs. We found this effort to be straightforward: the photo gallery contains 592 lines of HTML and JavaScript, of which only about 10 lines deal with camera access; we added video streaming functionality in 20 lines of code.

As another experiment, we have extended our photo gallery to support offline operation by utilizing two additional ServiceOS resources: local storage and network connectivity detection. The gallery first registers to receive asynchronous connectivity events. If it learns of network disconnection, it queues any new photos taken into local storage, and transparently synchronizes them later when it receives a "connected" event[5]. This offline functionality took only 40 lines of JavaScript to implement.

# 8 Related Work

Our work applies experience from decades of operating system research to the browser design. In this section, we first compare and contrast with existing browsers, and then discuss relevant resource management literature.

## 8.1 Web Browsers

MashupOS [41] first identified browsers as a multi-principal execution environment where the principal is a web site. MashupOS proposed programmer abstractions that enable browsers to offer the protection and communication model of a multi-principal OS. MashupOS addresses neither the re-architecting of browsers as an OS nor issues in resource management.

Recent work looked at re-architecting the browser with better protection. IE 8 [20], Google Chrome [32, 4], and OP [14] are all multi-process browser architectures. What fundamentally sets the Gazelle browser [42] apart from the aforementioned browsers is that Gazelle has a multi-principal OS construction which gives its browser kernel the *exclusive* control of cross-principal protection. In contrast, the other browsers' protection logic spreads into the principal space. The authors of Gazelle gave a comprehensive comparison of its browser architecture with the others. ServiceOS largely adopted Gazelle's architecture. While Gazelle addressed only the protection architecture, we tackle resource management for web applications. All browsers to date don't handle resource management, which is a significant missing piece in browser design. Reis et al advocated relying on the operating system's resource management for Google Chrome [32]; we have shown that the OS's resource management can be ill-suited for web applications (Section 1 and Section 7).

[5]Our renderer supports a new scheme, `localstorage://`, which allows web sites to display images from their local storage.

## 8.2 Access Control

Unix-based OSes use file-system-based access control. Individual physical devices are mapped to files and the permissions are set accordingly. Drivers, as well as higher-level software that directly accesses devices, directly manipulate access control defaults for each physical device. Windows-based OSes use access tokens to describe the privileges of a user account, and use security descriptors to describe the access control list for a securable object. A securable object can be a named Windows object, such as files and physical devices, or an unnamed object, such as process or thread objects.

While it is possible for a browser's access control design to directly use Unix or Windows-style access control by mapping web site principals to distinct user principals, such an approach is not without drawbacks. First, it sacrifices the browser's cross-platform portability and ties the browser to a particular OS's security model. Second, Unix and Windows require configuring access control for each physical device, including newer models of an old device and devices of the same semantics. This complicates adding and upgrading new devices. Finally, browsers often do not have sufficient privileges to modify system-wide device security policies; granting them this power would pose an unnecessary security risk.

In contrast, ServiceOS's access control design is *OS-independent*, *semantics-aware* (access control is tied to semantics of a class of resources, rather than physical devices) to ease extensibility, and *flexible* with various access control granularities. We strictly separate access control policy from actual resource access for a simple and robust system.

Some systems have defined applications [44, 12] as principals and may even use the digital signatures of mobile code providers [40] for labeling principals. In contrast with this work, ServiceOS treats a web site or web application as first-class principal for all the tasks in the operating system, including not only access control but also resource sharing and protection.

Android [12] applications declare desired access rights in a manifest, which is approved by the user during an application's installation. The manifest uses a flat namespace for all access rights. ServiceOS also allows web ap-

plications to use manifests to request a group of access rights, but unlike Android's flat namespace for capabilities, our design uses the Resource Object Model to hierarchically organize resource functionality according to resource semantics. This benefits extensibility, enables coarse-grained access control which simplifies manageability by reducing the number of access rights that need to be granted, and makes it more convenient to expose resources to web applications using the DOM.

**Browser plug-ins.** Existing browsers don't manage resources, and they expose the underlying OS to browser plugins. Therefore, some plugins, such as Java [40, 7], Flash, or Google Gears [13], have implemented parts of resource management, such as controlling access to webcams, in their own runtimes. This practice is flawed in that many plug-in security policies coexist, and often conflict, within the same browser. Worse, the compromise of *any* plug-in [36] allows a malicious web site to bypass the plug-in security model and access any devices at will. In contrast, our model unifies resource management for all components of a modern browser and does not depend on any support from any of the components.

## 8.3 Resource Sharing

In ServiceOS, we build on many existing proportional scheduling *mechanisms* [39, 5, 3] to enable our DOM-recursive *policy*. Lottery scheduling [39] pioneered such a mechanism and advocated separation of scheduling policy from scheduling mechanism. They experimented with a load insulation scheduling policy for separating one user's tasks from another's. Linux has a group scheduler [5], which is a CPU scheduling mechanism that enables fair scheduling across user principals; e.g., a user with 10 processes and another user with one process can share the CPU fairly, rather than allowing the first user to receive $10/11$ of the CPU. Resource containers [3] are proposed as a more fine-grained unit of resource allocation than a process; a single web server process can use one resource container for processing each incoming request, allowing resource sharing within a process. Hierarchical resource containers can be used to recursively allocate resources down the container hierarchy.

We can use any of these scheduling mechanisms to realize our DOM-recursive resource allocation policy; we used lottery scheduling in our implementation due to its simplicity and flexibility. Our contribution is formulating a policy for web browsers, where it is necessary to use the principal embedding hierarchy as the default policy for scheduling CPU, bandwidth, and transactional resources.

The hierarchical CPU scheduler in [28] is designed to support a variety of hard and soft real-time, as well as best-effort applications on a general-purpose operating system. The nodes in its hierarchy represent application classes with different QoS requirements, while the leaves are schedulable threads. This hierarchy is entirely different from our principal embedding hierarchy, where all the nodes are schedulable, and the hierarchy partitions resources among all nodes in the tree.

Regehr [31]'s hierarchical loadable scheduler (HLS) consists of a hierarchy of heterogeneous, independent schedulers, each of which may use a different scheduling algorithm for the purpose of real-time systems. HLS matches each application request with a scheduler that can meet its requirements. Our policy only requires a homogeneous, proportional scheduling algorithm.

# 9 Conclusions

Resource management has been a significant missing piece in browser design to date. In this paper, we presented our design for resource management for web applications in ServiceOS, a multi-principal OS where a web site is a first-class OS principal. ServiceOS provides web applications with systematic and consistent access and control of common I/O devices, such as cameras or microphones, by centralizing access control policies in an extensible layer. For sharing divisible resources, such as CPU and network bandwidth, ServiceOS uses a DOM-recursive sharing policy by default when in resource contention. This allows ServiceOS to insulate resource usage of a web page from untrusted content it embeds, and to allocate resources fairly among multiple applications embedded on the same web page. We also introduce *application-specified* resource allocation to allow web programmers to explicitly influence resource allocations based on web application semantics.

We have built and evaluated a ServiceOS prototype that manages a wide range of resources. Our evaluation shows that compared to existing browsers, ServiceOS provides web applications with superior service quality, fairness, and security.

# References


[1] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *Proc. 7th Intern. Workshop on Experimental Algorithms (WEA)*, volume 5038, pages 208–219, Provincetown, USA, 2008.

[2] J. P. Anderson. Computer security technology planning study. Technical report, Deputy for Command and Management Systems HQ Electronic Systems Division, October 1972.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server

systems. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, Berkeley, CA, 1999.

[4] A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The security architecture of the chromium browser, 2008. http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[5] CFS group scheduling, July 2007. http://lwn.net/Articles/240474/.

[6] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, 1994.

[7] G. Czajkowski and T. von Eicken. JRes: a resource accounting interface for Java. *SIGPLAN Not.*, 33(10):21–35, 1998.

[8] Document Object Model (DOM), January 2005. http://www.w3.org/dom/.

[9] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2008.

[10] M. A. Eriksen. Trickle: A Userland Bandwidth Shaper for Unix-like Systems. In *Usenix*, 2005.

[11] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, Berkeley, CA, 1994.

[12] Google Android. http://code.google.com/android/, 2008.

[13] Google Gears. http://gears.google.com/, 2008.

[14] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Securiy and Privacy*, 2008.

[15] G. Hamilton and P. Kougiouris. The spring nucleus: a microkernel for objects. In *Usenix-stc'93: Proceedings of the USENIX Summer 1993 Technical Conference on Summer technical conference*, pages 1–15, Berkeley, CA, USA, 1993. USENIX Association.

[16] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, New York, NY, 1992.

[17] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Proceedings of the Web 2.0 Security and Privacy Workshop (W2SP)*, May 2010.

[18] HTML 5 Editor's Draft, October 2008. http://www.w3.org/html/wg/html5/.

[19] G. Hunt and J. Larus. Singularity: Rethinking the Software Stack. In *Operating Systems Review*, April 2007.

[20] What's New in Internet Explorer 8, 2008. http://msdn.microsoft.com/en-us/library/cc288472.aspx.

[21] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, New York, NY, 2007.

[22] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the development of application-specific virtual memory management. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, New York, NY, 1993.

[23] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[24] D. Mcnamee and K. Armstrong. Extending the mach external pager interface to accommodate user-level page replacement policies. In *In Proceedings of the USENIX Association Mach Workshop*, pages 17–29, 1990.

[25] Mobile Device APIs Style Guide. http://www.openajax.org/member/wiki/Mobile_Device_APIs_Style_Guide, September 2008.

[26] A. Moshchuk, S. D. Gribble, and H. M. Levy. Flashproxy: transparently enabling rich web content via remote execution. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, New York, NY, 2008.

[27] NetLimiter, March 2009. http://www.netlimiter.com/.

[28] X. G. Pawan Goyal and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. pages 107–121, Seattle, WA, Oct. 1996. USENIX Assoc.

[29] D. Petrou, J. W. Milford, and G. A. Gibson. Implementing lottery scheduling: matching the specializations in traditional schedulers. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, 1999.

[30] A. Popescu. Geolocation API Specification. http://dev.w3.org/geo/api/spec-source.html, November 2008.

[31] J. D. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems*. PhD thesis, University of Virginia, May 2001.

[32] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of Eurosys*, 2009.

[33] F. Reynolds. An architectural overview of alpha: A real-time, distributed kernel. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 127–146, Berkeley, CA, USA, 1992. USENIX Association.

[34] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Computer Communications Review*, 27(1):31–41, 1997.

[35] J. Ruderman. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html.

[36] Secunia. Adobe Flash Player multiple vulnerabilities. http://secunia.com/advisories/26027/, July 2007.

[37] A. S. Tanenbaum, S. J. Mullender, and R. V. Renesse. Using sparse capabilities in a distributed operating system. In *Proc. of the Sixth IEEE International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[38] V8 JavaScript Engine. http://code.google.com/p/v8/, February 2009.

[39] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, Berkeley, CA, 1994.

[40] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. *SIGOPS Oper. Syst. Rev.*, 31(5):116–128, 1997.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions in MashupOS. In *ACM Symposium on Operating System Principles*, October 2007.

[42] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the $18^{th}$ USENIX Security Symposium*, Montreal, Canada, August 2009.

[43] B. Wilburn, M. Smulski, K. Lee, and M. A. Horowitz. The light field video camera. In *in Media Processors 2002*, pages 29–36, 2002.

[44] T. Wobber, A. Yumerefendi, M. Abadi, A. Birrell, and D. R. Simon. Authorizing applications in singularity. *SIGOPS Oper. Syst. Rev.*, 41(3):355–368, 2007.

[45] Xtreme Labs. What is the real speed of your 3G connection? http://www.xtremelabs.com/blog, December 2008.