CSE 221: Homework 2

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

1 GMS Design

1.1 Why did GMS chose this restriction? What problem does it solve?

The design of global memory is to guarantee any page reside in global memory pool is resistant to system crash. To be more specific, if there is a system crash, all global memory is reconstructable from disk, and there will be no information loss. However, to pursue this mechanism, one needs to make sure that the page in global memory is synchronized with disk, or said *clean*.

1.2 Describe how you might change GMS to safely allow it to write dirty pages to the global memory.

According to the definition of *dirty page*, it means that page has been edited but not be synchronized by others yet. That is to say, allowing dirty pages in global memory, to some certain, breaks the motivation of global memory management. To prevent potential issues, one possible solution is to do the clean-up procedure for every *dirty page* added into global memory.

1.3 Why does GMS require trusted nodes? What risks would be exposed by an untrusted node?

In the view that most needed information is provided by nodes. Fake information from untrusted nodes will be very dangerous to the whole system. Two possible scenarios are described as follows.

- GMS believe each added page is a clean page. If this rule is violated, some pages in global memory will never ne constructed when a fault occurs.
- GMS swap pages based on the age information provided by nodes. If a malicious node consistently says it is younger than others, when a page swap is required, other nodes will be swapped out instead. Thus, that malicisous node affects the whole system.

1.4 Describe how you might change GMS to work with untrusted nodes?

Since some nodes might be malicious, the system must decrease the reliability of individual nodes. For example, system should not fully rely on the information given by nodes. Instead, it should check the page content to verify whether it is clean or not and monitor the age of each page by itself.

Another more troublesome solution is to derive a mechanism to verify whether each node is trusted or not. For those verified nodes, it can work as usual. On the other hand, for those untrusted ones, it should provide a more restricted set of services.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

2 Transparency vs Optimization

2.1 Sprite

2.1.1 (1)

Sprite decides to hide the information from users, or said, it pursues transparency. They implement the file system as a collection of domains on different servers but share one same namespace, so that each single user will not discover he is actually accessing file remotely on a distributed system.

2.1.2 (2)

What it plans to hide from users include the follows.

- Namespace on different machines is hidden. It uses a single uniform namespace, so that all files and devices on all machines are accessible anywhere in network. To achieve this, prefix table mechanism is used.
- Memory usage is hidden. It shares memory for different processes for the appearance of multiprocessors, which effectively increases the system performance.
- The exact process execution location is hidden. It will migrate a process to a idle machine if it is needed.
- The exact write execution timing is hidden. It uses delayed writes to increase the system performance.

2.1.3 (3)

To preserve the location of each data, a prefix table mechanism is developed. The table is used to locate whether a file is locally available or it can be a remote node on the cluster.

2.1.4 (4)

To fulfill the cache mechanism for delayed writes, it must make sure of the cache consistency. An error might happen when a cached information is not updated on time. To solve this issue, Sprite must make sure updates are propagated to other system right after certain editions are made.

2.1.5 (5)

In certain level, the authors successively built up a system with every goal they pursued. However, this system will encounter increasing burden once the cluster gets larger. In other words, I believe this system is not good at scaling up.

2.2 Xen

2.2.1 (1)

The goal of Xen is to build up a high-performance virtual machine monitor that supports strict resource control among multiple guest OSes. Thus, for each guest OS, the implementation and resource is hidden from Xen.

2.2.2 (2)

What Xen hides from guests include the follows.

- Memory Virtualization. Xen helps to manage memory and deal with memory page faults, instead of each guest OS. On top of that, it allows hardware to access guest OS table directly.
- CPU Virtualization. Xen explicitly schedule CPU for each guest OS. In other words, guest OSes are no longer run at highest privilege (using ring 1 instead of ring 0).
- Storage Virtualization. Xen reads from and writes to a virtual disk, instead of a real one.
- Network Virtualization. Xen implements the socket sent out from guest OSes by Ethernet.

- Display / IO Virtualization. Xen virtualizes every input device, such as keyboard and mouse, and output device, for example, monitor display.

2.2.3 (3)

To virtualize CPU, Xen is responsible to swap in and swap out guest OSes. Thus, it needs to maintain the real running time used by each guest OS. On the other hand, guest OS uses CPU as normal and cannot tell the difference from running on a real CPU.

2.2.4 (4)

In order to virtualize exception handler for guest OSes on Xen, Xen allows each guest OS to register their own exception handlers. By doing so, when an exception occurs on a specific guest OS, Xen will pass the exception directly to that OS. However, it is possible that a non-handled exception is raised on that OS, which is called "double faults" problem. To solve this, Xen will directly trap these error by itself.

2.2.5 (5)

In my opinion, the authors also achieved their goal. The only problem is caused by the paravirtualization, which requires certain modifications on guest OS first. In other words, this part of information is not perfectly hidden.

2.3 Grapevine

2.3.1 (1)

Grapevine, on the other hand, pursues the optimization. It basically implements two services: message and registration, both of which do not hide anything from users.

2.3.2 (2)

For message service, system accepts any message for delivery.

For registration service, it not only uses a table to map each user name to user information, used machine, locations, organization, and applications, but also ask for password authentication.

2.3.3 (3)

Registration servers know the names, addresses of all messages as well as registration servers, names of all registries and servers that contain replicates, so that it can use an algorithm to locate the closest servers on a local Ethernet.

2.3.4 (4)

To prevent the scale of computation from growing as a function of system size, it derives two solutions.

- It uses distribution lists, so that the services work like several groups.
- It partitions the whole system into two-layered hierarchy.

2.3.5 (5)

In my understanding, I believe the authors perfectly created a system with no hidden information.

3 Lazy Evaluation

3.1 Mach

3.1.1 (1)

In Mach, it uses *shadow objects* to achieve lazy evaluation.

3.1.2 (2)

Shadow object is implicitly implemented in Mach kernel, which utilizes the concept of copy-of-write.

3.1.3 (3)

Shadow object is a kind of memory object, which records only the modified pages among a shared memory segment owned by multiple processes. If one of these processes writes into one shared page, *shadow object* records only the modified one, so that copy-on-write mechanism can be fulfilled and prevent duplicate memory.

Copy-on-write is clearly a kind of lazy evaluations, since the rest of the memory does not need to be copied right after the edition. It has been a prevalent way to deal with memory management.

3.1.4 (4)

Copy-on-write is an efficient strategy with the assumption that most of the memory shared by multiple processes will not be changed. In other words, if this assumption fails, or said an abundant of editions are made from multiple processes to multiple shared pages, copy-on-write mechanism will not help anymore.

3.2 Exokernel

3.2.1 (1)

Exokernal uses *secure binding* to achieve eager evaluation.

3.2.2 (2)

Secure binding is implicitly implemented in Exokernel, which takes the advantage of one-time resource binding strategy.

3.2.3 (3)

In Exokernal, library OSes are responsible for the resource management; whereas the kernel is used for protection control. In the implementation, every process needs to be verified only once, which effectively saves the checking time and significantly improves the overall system performance.

3.2.4 (4)

Secure binding has a very strong assumption that most of the processes will use few resource, and thus, few verified resources are used repeatedly. That is to say, once this suggestion fails, or said, each process utilizes a huge amount of resources for only few times, benefits gain from eager evaluation is not obvious anymore.

3.3 Xen

3.3.1 (1)

Xen's *safety check* utilize lazy evaluation.

3.3.2 (2)

Safety check is implicitly implemented in Xen system.

3.3.3 (3)

The only required check is that the handler's code segment does not specify execution in ring 0, since other problems are solvable during exception propagation. Thus, lazy evaluation is applied.

3.3.4 (4)

The lazy evaluation has a strong assumption that there is no malicious OS, which is too strong to be true. To be more specific, malicious users might register very high amount of exception handler, and thus, greatly slow down the Xen performance.

3.4 VAX / VMS

3.4.1 (1)

VAX/VMS delays cleaning up a physical page, which is a kind of lazy evaluation.

3.4.2 (2)

Delay in page cleanup and preparation is implicitly implemented in VAX / VMS, which saves the time when a fresh page is required by a process.

3.4.3 (3)

When a fresh page is required by a process, it might not be so important to prepare the available page right away. Instead, a fake page can be created and returned first. Later, when that page is really needed, a trap will occur, and OS can remap a real physical page with empty contents.

3.4.4 (4)

The lazy of page initialization is based on the assumption that new created page will not be used directly. That is to say, once this assumption is not guaranteed anymore, or said, read/write to new created page are common pattern, delay in page cleanup and preparation will not be so beneficial.