Xkvisor

Bryan Yue, Omeed Magness, Sandip Samantaray, Edward Matsushima, Bernard Kim
University of Washington
{byue, omag01, sandip80, edwarm4, kimb0128}@cs.washington.edu

ABSTRACT

This paper introduces *xkvisor*, an extension to the monolithic, POSIX-like, teaching operating system xk [1]. Xkvisor defines the interface to allow customizable, lightweight guest operating systems[1] to be run from user mode.

In this paper, the focus is on two features of xkvisor that we have implemented[2]. The first feature is the virtual memory (VM) interface, which exposes a way for a guest operating system to request physical pages from xkvisor and interface with the application's page table. The guest OS is expected to use this interface to create and manage other user process within its isolation domain. The second feature we discuss is system call redirection, which allows a system call from a guest OS user application delivered to xkvisor (via the normal `syscall` instruction) to be forwarded to and handled by a guest OS running as a background process. Importantly, these system calls can be handled by the guest OS asynchronously via the use of a syscall buffer and sleeping application processes.

These features are made possible by adding the notion of "privileged system calls" (which are made by a guest OS to xkvisor to request physical resources) and a "universal syscall," which the guest OS user process can use to make system calls to xkvisor that redirect to the corresponding guest OS. We've found that both features provide a reasonable way to run a simplistic guest OS, which creates a user process through the VM interface, and then responds to syscalls from that user process via the syscall redirection mechanism.

1. INTRODUCTION

Many operating systems in use today impose strict abstractions on user processes running on that system, such as fixed ways of doing inter-process communication, file I/O, etc. For some operating system designs (e.g. monolithic), this can cause some (or all) of those high-level abstractions to be implemented in the kernel, which can make the kernel a performance bottleneck, and certainly will increase kernel code size and thus decrease confidence in kernel correctness. It would be ideal to push some of these abstractions up to user space, because this would mean less code is running in full CPU privileged mode. Some kernel designs already do this to some extent (e.g. microkernels).

In a monolithic kernel, these fixed abstractions decrease customizability from the user perspective. It can sometimes be beneficial for an application to have direct access to hardware features or exposure to a different abstraction, or different level abstraction. For example, some performance-critical applications (e.g. databases, language runtime garbage collectors) can make optimizations if given low-level hardware access, and other applications might prefer still high-level, but slightly different, abstractions (e.g. a Windows application running on Linux looking for the Win32 API).

Xkvisor tries to solve this problem by pushing all the high-level operating system functions into a guest OS, and, from the host, only exposing a very low-level API that focuses on mediating access to the hardware, allocating and isolating physical resources, and redirecting communication between the guest OS and its applications. This leaves the management policies and higher-level abstractions for the guest to implement in user space. The goal is to simplify the interface, allow customizability, and limit machine-dependence as much as possible.

This paper focuses on the virtual memory interface and system call redirection in xkvisor. Section 2 will provide some background on microkernels, since our design is a modified microkernel design. Section 3 presents a detailed explanation of our design and implementation. Section 4 discusses challenges we ran into during implementation. Section 5 presents results of this project

[1] A "lightweight guest OS" is similar to a guest OS in the virtual machine setting because it can start and manage processes, but different in that it is not a goal for the guest to be able to run on standalone hardware. Our guest OS is built on top of some high-level host OS abstractions and is more like a container (isolated collection of applications) where the (syscall) interface exposed to the applications inside the container is defined by the container.

[2] xkvisor is a group undergraduate research project led by Professor Tom Anderson and graduate students Danyang Zhuo and Matthew Rockett. The group is broken into multiple subgroups that are working on different parts of the project (e.g. one group working on user-level threads, one on the disk interface, etc.). This paper will focus on the subgroup of the authors of this paper, which is the virtual memory system.

(primarily design results). Section 6 mentions related work. Section 7 concludes.

2. BACKGROUND ON MICROKERNELS

In a microkernel, the goal is to move functionality to user space to minimize the kernel. The kernel retains only low-level abstractions, and the rest of the kernel is now a collection of user-level services, grouped by function (e.g. a file system process, network stack process, graphics handling process, etc.). Microkernels can also expose multiple APIs, as in the case of Windows NT (which provides OS/2, POSIX, Win32, etc. APIs).

Microkernels handle system calls by forwarding to the appropriate user process. For example, the kernel might forward a file write system call to its user space "file system process."

Microkernels also provide many benefits over traditional monolithic kernels. As mentioned earlier, the kernel code base (code running in privileged CPU mode) can be smaller, because some work is being delegated to unprivileged, user-level processes. Multi-level (multi-API) microkernels also provide some amount of customizability because user processes can pick the appropriate API from a couple options.

Xkvisor's design is like a microkernel, except that it doesn't have multiple user-level processes. For now, the guest OS is a single process that just supports a few simple system calls. Once the xkvisor project is fully integrated across subgroups, it is likely that there will be multiple user-level processes representing a guest OS.

Additionally, xkvisor does not limit the user-facing interface to be a few built-in choices. Xkvisor exposes a low-level API that any custom guest OS can be implemented on top of, and that guest can expose the abstractions it would like.

3. DESIGN AND IMPLEMENTATION

In xkvisor's interface, guest operating systems are each simply a generic user-level process. What this means is that there is no existing abstraction in xk (which has a limited POSIX interface) for a user-level process to be managing another process' page table. To add support for this, we introduced *privileged system calls*, system calls which expose a lower-level abstraction than a typical syscall, and which are meant only for a guest operating system to call.

This is enforced by a special bit in the process metadata to indicate that process is a guest OS and can make privileged system calls. Our convention is that we preface privileged system call names with 'g' to indicate that they are intended for a guest OS to call.

We will now discuss two features of xkvisor: the virtual memory system (resource allocation), and system call redirection.

3.1 Resource Allocation

At a high level, for a guest OS to run a guest user process, it needs to create a process, map some memory into that process's address space, and schedule the process.

This requires some privileged instructions (e.g. `mov` to `%cr3` to load in a page table of a process it is attempting to schedule), so xkvisor exposes these privileged steps as privileged system calls.

Xkvisor uses two key data structures to track which guest operating systems have control of specific resources.

One key data structure is the physical memory map inside of each guest OS's process metadata (recall that each guest OS is just a user process). This is a map of physical page number to page state – 0 for not owned, 1 for owned, or 2 for owned and in use by an application. The physical page number (PPN) is the index into the physical memory map. A guest OS can get back a copy of this data structure from the host OS via a `gquery_user_pages` privileged system call.

The second data structure xkvisor uses is a process table map. This is just a mapping from process id to a memory segment struct, which indicates if a user process is owned by a guest OS, and if so, what the base, midpoint, and bound virtual addresses are (used for the stack and heap virtual address boundaries).

There are two copies of each of these maps. One copy is for the guest OS to modify/read, and one copy is inside xkvisor for security purposes. Although our privileged system calls that modify xkvisor's maps will also modify the guest OS maps via a return parameter, it is the responsibility of the guest OS to properly manage its copy of these maps.

We currently abstract xkvisor's underlying host process structs as process IDs for the guest OS to use when creating a user application process. This is obviously not ideal and will be reworked once we implement fork and exec at user level and expose privileged system calls for the guest OS to control scheduling itself. In future work, the guest OS will have its own representation of a process and will make a privileged system call to schedule this in.

Given these data structures, a guest OS can create a process with the privileged syscall interface:

```
// Get a process from the host OS with the
// given virtual address description. Updates
// the guest OS's process map as well. Returns
// a pid for the new process.
int grequest_proc(struct app_va_segment
*proc_map, uint64_t base, uint64_t midpoint,
uint64_t bound);
```

```
// Loads the executable ELF from the given file
// into the given process's code segment.
int gload_program(int pid, char *path);

// Finishes setting up a process with the given
// arguments (primarily puts the args on the
// process's user stack).
int gdeploy_program(struct syscall_message
*args);

// Returns the process with given pid back to
// the host OS for reclamation.
int greturn_proc(int pid);


// Copies the host's physical memory map to the
// guest OS stack.
int gquery_user_pages(uint8_t *page_map);

// Adds a mapping in the given process'
// page table to point to the given physical
// page with the given flags.
int gaddmap(int app_pid, int host_ppn, uint64_t
va, int app_present, int app_writeable);

// Removes the mapping in the given process'
// page table with the given virtual address.
int gremovemap(int app_pid, uint64_t va);


// Updates the flags on the existing mapping in
// the given process' page table.
int gupdate_flags(int app_pid, uint64_t va, int
app_present, int app_writeable);
```

The typical workflow a guest OS would go through to create a process looks like the following:

1. A call to `grequest_proc`, where a guest OS passes in its copy of the process map. This is a little awkward and will be fixed in future version, but for now, both the guest OS and host keep a copy of the process state map discussed earlier, and it is the responsibility of the guest OS to maintain it. In addition to the process map, the guest OS also provides a base virtual address, midpoint, and bound, and the host OS creates a process in its own process table to represent the guest user process (with the given virtual address bounds).
2. A call to `gload_program` to load the executable (ELF) code from a given file on disk into a given guest user process code region.
3. A call to `gdeploy_program` to finish setting up a process with the given arguments. The process will be scheduled after a call to `gresume` sets the process to be "runnable" (see section 3.2)

At any point during the lifecycle of a process, the guest OS uses `gaddmap`, `gremovemap`, and `gupdate_flags` to update the page table of its guest user process (for now, all page table updates are done individually through this interface, but eventually we want to batch these updates). All page tables are currently kept in kernel memory, so these privileged syscalls are necessary for a guest OS to modify a page table.

Xkvisor will check that each of these attempted page mappings are operating on physical pages belonging to the guest OS, as determined by the page map in the process's metadata (note how xkvisor is not doing any memory management logic for the guest process, just verification that the operation is legal). These calls must be used to add pages to a guest user process' stack and heap, which start out empty.

Because xkvisor verifies all page table operations from the guest OS, it's safe for guest user process page tables to map from virtual to physical (i.e. no Vt-X "Extended Page Tables" or shadow page tables are needed). Xkvisor is mapped in the top of each guest OS's address space (user process and guest OS) at address `KERNBASE`, but guest operating systems are not mapped into their user processes, since both the guest user process and guest OS are at ring 3 (lowest privilege level), and this would require special care to protect guest OS memory.

In order for the guest OS to be able to access user process memory, user process stack/heap pages are mapped in both the guest OS and user process page tables at the same virtual address for the same physical pages (Fig. 1a, Fig. 1b). The stack/heap boundary is set at 3G, with the heap growing up to the bound value of 4G, and the stack growing down to the base value of 2G. The guest OS can specify these boundaries when the user process is being initialized. With several user processes per guest OS, this design is not viable since the maximum memory usage for the user process is not known ahead of time and having set boundaries for the virtual address space is not flexible. Adjusting this design is future work.

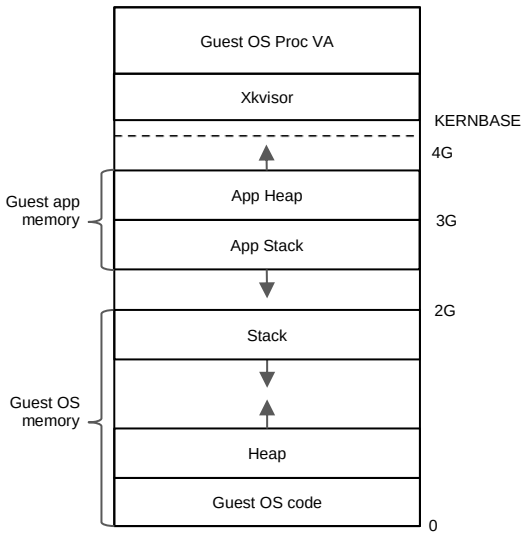

Fig. 1a: Guest OS Virtual Address Space

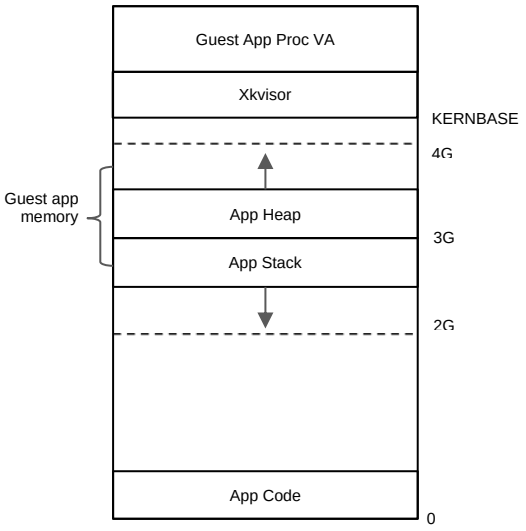

Fig. 1b: Guest OS User Process Virtual Address Space

When a guest OS is done with a process, it can return it to xkvisor to reclaim the resources via the `greturn_proc` call.

For now, each guest OS is statically assigned a certain number of physical pages to allocate for use in user applications. Eventually we want to allow the guest OS to request for more physical pages or return extra pages.

3.2 System Call Redirection

The other major feature we will discuss in this report is system call redirection, through which a guest user process can make a system call to another background guest OS.

Syscalls are a bit tricky to handle synchronously because both the guest user and guest OS are just normal xkvisor processes, which can be interrupted any point. We want syscalls to be asynchronous, but also need to guarantee that a guest user process that makes a syscall is not rescheduled before its guest OS has finished handling the syscall. We achieve this by putting the current process to sleep on a syscall and guarantee that it is not awoken until its corresponding guest OS handles the system call.

There is also the question of how to communicate the syscall information if syscalls are asynchronous. Our solution was to make a struct representing a syscall, which holds the syscall index (defined by the guest OS), arguments, current proc pid and pointer to the next struct. Xkvisor will add one of these new syscall "message" structs to the end of a linked list of structs for a guest OS, and the guest OS can read from the front and handle a syscall whenever it is scheduled in.

Syscall redirection is implemented by the following:

On guest syscall:
1. Wake up guest OS process
2. Add a syscall message to its syscall buffer
3. Put guest user process to sleep

When guest OS is rescheduled:
1. Reads the next entry from its syscall message buffer or sleeps (atomically) if there are no syscalls
2. Handles a syscall
3. Wakes up originating user process

This means that the main part of the guest OS can just be the simple while loop:

```
while (1) {
  gnext_syscall(…)
  // handle syscall
  gresume(…)
}
```

With these two privileged system calls:

```
// Consumes the next system call from the
// kernel process metadata and places it in the
// given syscall message OR puts this process
// to sleep if there are no syscalls waiting to
// be handled.
int gnext_syscall(struct syscall_message
*message);

// Sets the process with given pid to runnable
// (so it will be scheduled again).
int gresume(int pid);
```

Note that it is critical for `gnext_syscall` to atomically put the guest OS to sleep if necessary, otherwise there could be dirty read concurrency issues.

As an optimization, rather than having xkvisor being aware of each of guest OS' syscalls (e.g. awrite, aread, afork, etc.), we made a universal syscall meant for the guest OS and then it is up to the guest OS to determine which syscall was intended through the syscall index:

```
int app_syscall(int syscall_index, struct
arg*);
```

So to walk through an example of the control flow of syscall redirection for two apps, "guest test" (guest user process) and "guest os":

1. User application process makes a write system call intended for the guest OS with `app_syscall`(WRITE, args)
    a. Trap into xkvisor
    b. Add syscall message to guest OS buffer
    c. Put guest user process to sleep
    d. Wake up guest OS

    e. Yield to scheduler; eventually get rescheduled in guest OS
2. Guest OS does a `gnext_syscall`
    a. Trap into xkvisor
    b. Xkvisor consumes syscall message and copies to guest OS (the '`awrite`'),
    c. Return to guest OS
3. Guest OS handles `awrite`
4. Guest OS does a `gresume` on user application
    a. Trap into xkvisor
    b. Xkvisor sets user application process to "runnable"
    c. Return to guest OS
5. Guest OS does a `gnext_syscall`
    a. Trap into into xkvisor
    b. Syscall message buffer is now empty, so xkvisor puts guest OS to sleep
6. User application process eventually gets scheduled and continues executing, with its syscall completed

Note that the significance of this redirection is on step 3, where the guest OS can now implement `awrite` however it wants, and the code for write can be removed from xkvisor, shrinking the size of the kernel.

4. CHALLENGES

During the implementation of these two features of xkvisor, we ran into many challenges along the way. We originally spent a while creating a great design and were trying to implement xkvisor in big chunks. When we attempted to do so, there was too many tasks to complete without a clear direction and it was tough to get runnable code.

After a suggestion from one of our research mentors, we decided to work on xkvisor incrementally. With this new strategy, we intentionally began with simpler designs and gradually tried to scale up towards our final design. As a result, we have a working version, but it is not the original design we wanted. We learned that this is the best possible way to go, because details in the design will change based on new challenges that are faced, and having a runnable version is important for providing directions for next steps.

Besides gaining more technical knowledge of microkernels, the most important takeaway was learning the iterative design approach and taking small steps towards the goal. For example, when dealing with deadlock issues between the shell/user application/guest OS processes, we decided to first handle just the guest OS and shell, and once the shell worked, we added the application process and resolved additional deadlock issues that arose. Implementing in large chunks ended up being terrible for momentum, correctness, and productivity.

5. RESULTS

The main result we got out of this project was the design of what we think is a reasonable virtual memory interface for a container/guest OS to use to create and communicate with its own guest user processes. We made many simplifications along the way but think that our prototype implementation also demonstrated that there was promise in our design.

In particular, we believe we've shown that asynchronous redirection of syscalls with sleeping/waking processes is viable via a simple interface and that a physical memory map page leasing scheme is reasonable as well. We have pushed some of memory management logic to the guest OS and will continue to transfer functionality from xkvisor to the guest OS.

While xkvisor's design can still be improved a lot, we've laid the groundwork for a simple interface with all the essentials needed for the guest OS to implement management logic tailored to the application's specific needs. As the design continues to evolve, additional features will be added if necessary, but simplicity will still be a goal as most of the management logic should be implemented in the guest OS.

6. RELATED WORK

For xkvisor, we heavily built off ideas from Exokernel, trying to push all the high-level operating system functions into a guest OS, and making the operating system expose a very low-level API that focuses on just protecting (but not managing) the hardware. Although our project is inspired by Exokernel, we have made many simplifications to it, such as abandoning the mechanism for a library OS to download code into the kernel, and not forwarding interrupts to a guest OS.

We also are using some ideas from the Xen microkernel [2] – in particular, we are following the design of a introducing "hypercalls" (really privileged system calls for us) for a paravirtualized OS to use. We are also requiring all page table update operations to go through xkvisor. We have abandoned some of the more complex aspects of the Xen design, such as the circular buffers for data transfer (I/O rings) and the usage of x86 ring 1 to isolate the guest OS. In addition, we borrowed Xen's idea of having the guest OS be aware of the physical page numbers to avoid additional translations via shadow page tables.

7. CONCLUSIONS

In this paper, we presented xkvisor, an operating system which provides a guest OS abstraction to user processes and provides some privileged system calls for these guest OS processes to make in order to request resources. The VM interface and syscall redirection system demonstrate that these two features are viable even with a simple design.

In the future, we would like to add additional functionalities to this prototype by allowing more than one guest user process, allowing more than on guest OS, and providing a more robust VM interface. We'd also like to expose synchronization to user level and implement multiprocessing there to remove the dependency on the underlying xkvisor scheduler and process table. In addition, we want to implement page fault redirection via a trap message buffer (similar to our syscall buffer) to reduce the time spent transitioning between user and kernel mode. All these additions will be iteratively accomplished in small steps.

REFERENCES

[1] T. Anderson, D. Zhuo, M. Rockett. et. al. *xk ("Experimental Kernel").* University of Washington. 2017 - present.

[2] P. Barham, B. Dragovic, H. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In SOSP, 2003.

[3] D. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Opearting System Principles,* pages 251-266, 1995.