Operating system organizaton

Required reading: Exokernel paper.

Intro: virtualizing

One way to think about an operating system interface is that it extends the hardware instructions with a set of "instructions" that are implemented in software. These instructions are invoked using a system call instruction (int on the x86). In this view, a task of the operating system is to provide each application with a *virtual* version of the interface; that is, it provides each application with a virtual computer.

One of the challenges in an operating system is multiplexing the physical resources between the potentially many virtual computers. What makes the multiplexing typically complicated is an additional constraint: isolate the virtual computers well from each other. That is,

- stores shouldn't be able to overwrite other apps's data
- jmp shouldn't be able to enter another application
- one virtual computer cannot hog the processor

In this lecture, we will explore at a high-level how to build virtual computer that meet these goals. In the rest of the term we work out the details.

Virtual processors

To give each application its own set of virtual processor, we need to virtualize the physical processors. One way to do is to multiplex the physical processor over time: the operating system runs one application for a while, then runs another application for while, etc. We can implement this solution as follows: when an application has run for its share of the processor, unload the state of the phyical processor, save that state to be able to resume the application later, load in the state for the next application, and resume it.

What needs to be saved and restored? That depends on the processor, but for the x86:

- IP
- SP
- The other processor registers (eax, etc.)

To enforce that a virtual processor doesn't keep a processor, the operating system can arrange for a periodic interrupt, and switch the processor in the interrupt routine.

To separate the memories of the applications, we may also need to save and restore the registers that define the (virtual) memory of the application (e.g., segment and MMU registers on the x86), which is explained next.

Separating memories

Approach to separating memories:

- Force programs to be written in high-level, type-safe language
- Enforce separation using hardware support

The approaches can be combined.

Lets assume unlimited physical memory for a little while. We can enforce separation then as follows:

- Put device (memory management unit) between processor and memory, which checks each memory access against a set of domain registers. (The domain registers are like segment registers on the x86, except there is no computation to compute an address.)
- The domain register specifies a range of addresses that the processor is allow to access.
- When switching applications, switch domain registers.

Why does this work? load/stores/jmps cannot touch/enter other application's domains.

To allow for controled sharing and separation with an application, extend domain registers with protectioin bits: read (R), write (W), execute-only (X).

How to protect the domain registers? Extend the protection bits with a kernel-only one. When in kernel-mode, processor can change domain registers. As we will see in lecture 4, x86 stores the U/K information in CPL (current privilege level) in CS segment register.

To change from user to kernel, extend the hardware with special instructions for entering a "supervisor" or "system" call, and returning from it. On x86, int and reti. The int instruction takes as argument the system call number. We can then think of the kernel interface as the set of "instructions" that augment the instructions implemented in hardware.

Memory management

We assumed unlimited physical memory and big addresses. In practice, operating system must support creating, shrinking, and growing of domains, while still allowing the addresses of an application to be contiguous (for programming convenience). What if we

want to grow the domain of application 1 but the memory right below and above it is in use by application 2?

How? Virtual addresses and spaces. Virtualize addresses and let the kernel control the mapping from virtual to physical.

Address spaces provide each application with the ideas that it has a complete memory for itself. All the addresses it issues are its addresses (e.g., each application has an address 0).

- How do you give each application its own address space?

    - MMU translates *virtual* address to *physical* addresses using a translation table
    - Implementation approaches for translation table:
        1. for each virtual address store physical address, too costly.
        2. translate a set of contiguous virtual addresses at a time using segments (segment #, base address, length)
        3. translate a fixed-size set of address (page) at a time using a page map (page # -> block #) (draw hardware page table picture). Datastructures for page map: array, n-level tree, superpages, etc.

    Some processor have both 2+3: x86! (see lecture 4)

- What if two applications want to share real memory? Map the pages into multiple address spaces and have protection bits per page.
- How do you give an application access to a memory-mapped-IO device? Map the physical address for the device into the applications address space.
- How do you get off the ground?

    - when computer starts, MMU is disabled.
    - computer starts in kernel mode, with no translation (i.e., virtual address 0 is physical address 0, and so on)
    - kernel program sets up MMU to translate kernel address to physical address. often kernel virtual address translates to physical adress 0.
    - enable MMU

    Lab 2 explores this topic in detail.

Operating system organizations

A central theme in operating system design is how to organize the operating system. It is helpful to define a couple of terms:

- Kernel: the program that runs in kernel mode, in a kernel address space.
- Library: code against which application link (e.g., libc).
- Application: code that runs in a user-level address space.

- Operating system: kernel plus all user-level system code (e.g., servers, libraries, etc.)

Example: trace a call to printf made by an application.

There are roughly 4 operating system designs:

- Monolithic design. The OS interface is the kernel interface (i.e., the complete operating systems runs in kernel mode). This has limited flexibility (other than downloadable kernel modules) and doesn't fault isolate individual OS modules (e.g., the file system and process module are both in the kernel address space). xv6 has this organization.
- Microkernl design. The kernel interface provides a minimal set of abstractions (e.g., virtual memory, IPC, and threads), and the rest of the operating system is implemented by user applications (often called servers).
- Virtual machine design. The kernel implements a virtual machine monitor. The monitor multiplexes multiple virtual machines, which each provide as the kernel programming interface the machine platform (the instruction set, devices, etc.). Each virtual machine runs its own, perhaps simple, operating system.
- Exokernel design. Only used in this class and discussed below.

Although monolithic operating systems are the dominant operating system architecture for desktop and server machines, it is worthwhile to consider alternative architectures, even it is just to understand operating systems better. This lecture looks at exokernels, because that is what you will building in the lab. xv6 is organized as a monolithic system, and we will study in the next lectures. Later in the term we will read papers about microkernel and virtual machine operating systems.

Exokernels

The exokernel architecture takes an end-to-end approach to operating system design. In this design, the kernel just securely multiplexes physical resources; any programmer can decide what the operating system interface and its implementation are for his application. One would expect a couple of popular APIs (e.g., UNIX) that most applications will link against, but a programmer is always free to replace that API, partially or completely. (Draw picture of JOS.)

Compare UNIX interface (v6 or OSX) with the JOS exokernel-like interface:

```
enum
{
        SYS_cputs = 0,
        SYS_cgetc,
        SYS_getenvid,
        SYS_env_destroy,
        SYS_page_alloc,
        SYS_page_map,
```

```
        SYS_page_unmap,
        SYS_exofork,
        SYS_env_set_status,
        SYS_env_set_trapframe,
        SYS_env_set_pgfault_upcall,
        SYS_yield,
        SYS_ipc_try_send,
        SYS_ipc_recv,
};
```

To illustrate the differences between these interfaces in more detail consider implementing the following:

- User-level thread package that deals well with blocking system calls, page faults, etc.
- High-performance web server performing optimizations across module boundaries (e.g., file system and network stack).

How well can each kernel interface implement the above examples? (Start with UNIX interface and see where you run into problems.) (The JOS kernel interface is not flexible enough: for example, *ipc_receive* is blocking.)

Exokernel paper discussion

The central challenge in an exokernel design it to provide extensibility, but provide fault isolation. This challenge breaks down into three problems:

- tracking owner ship of resources;
- ensuring fault isolation between applications;
- revoking access to resources.

- How is physical memory multiplexed? Kernel tracks for each physical page who has it.
- How is the processor multiplexed? Time slices.
- How is the network multiplexed? Packet filters.
- What is the plan for revoking resources?
    - Expose information so that application can do the right thing.
    - Ask applications politely to release resources of a given type.
    - Ask applications with force to release resources
- What is an environment? The processor environment: it stores sufficient information to deliver events to applications: exception context, interrupt context, protected entry context, and addressing context. This structure is processor specific.
- How does on implement a minimal protected control transfer on the x86? Lab 4's approach to IPC has some short comings: what are they? (It is essentially a polling-based solution, and the one you implement is unfair.) What is a better way? Set up a specific handler to be called when an environment wants to call this

environment. How does this impact scheduling of environments? (i.e., give up time slice or not?)

- How does one dispatch exceptions (e.g., page fault) to user space on the x86? Give each environment a separate exception stack in user space, and propagate exceptions on that stack. See page-fault handling in lab 4.
- How does on implement processes in user space? The thread part of a process is easy. The difficult part it to perform the copy of the address space efficiently; one would like to share memory between parent and child. This property can be achieved using copy-on-write. The child should, however, have its own exception stack. Again, see lab 4. *sfork* is a trivial extension of user-level *fork*.
- What are the examples of extensibility in this paper? (RPC system in which server saves and restores registers, different page table, and stride scheduler.)