2019-09-04

Why is studying OS's interesting?

- Tradeoffs
  - Efficiency vs portability (or abstraction or convenience)
  - Power vs simplicity
  - Flexibility of use vs security
- Difficulty and uniqueness of task
  - You're writing bare metal C without the aid of any syscalls
- New hardware constantly opens up new opportunities for OS innovation

How is this course structured

- Lectures explore ideas in OS construction, the inner workings of xv6
- Labs exercises our understanding of the material
  - Labs are split between **system programming**, **os primitives**, and **os extras**
- Reading papers covers modern topics in OS construction
- There are **two** exams
  - Midterm (in class)
  - Final
- 6.828 vs 6.S081
  - 6.828 will be pure research topics in the future
  - 6.S081 will be the undergrad version that will be available in the future

Roles of operating systems

1. Abstract hardware features behind a simple interface
2. Multiplex/multitask programs
3. Isolation of programs from one another
4. Supply primitives for sharing between processes
5. Orchestration of sharing and isolation primitives in such a way that provides security and performance

Core ideas of developing operating systems

- **Abstraction** = the principle of
  - In operating systems, oftentimes abstractions are created that don't map neatly onto hardware features
    - e.g. a filesystem has concepts of permissions and distinctions between different types of data, but the disk is just a opaque binary storage

- **Kernel** = embodies the notion that the OS should be entirely separate from the programs that run on it
  - **User-space vs Kernel-space** = the idea that encapsulates this separation
    - User-space contains things like running programs, text editors, compilers, etc
    - Kernel-space contains things like process control, device drivers, file systems, and the code that directly talks to hardware
  - **System call interface** = the collection of syscalls that allow userspace programs to trigger desired functionalities in the kernel
    - **System call** = an instruction in program whose opcode does not exist in the CPU's ISA but does have meaning to an operating system
      - Recall from 6.004
    - Modern OS's have hundreds of system calls available
      - This class focuses on a much smaller core set that has existed for decades (*i.e.* POSIX)

xv6

- **xv6** = a UNIX-like OS that is small enough to understand completely
  - Borrows UNIX ideas like file descriptors
    - **File descriptor** = an integer that represents a process's ownership over a file or resource
      - 0 is always *stdin*
      - 1 is always *stdout*
      - A mapping between a proccess's file descriptors and their referent objects is maintained in a kernel table

Tale of two syscalls: fork and exec

- **fork** = a syscall that generates a complete copy of the execution state for the current process and then returns an integer that communicates which process follows
- **exec** = a syscall that overwrites the virtual address-space of the current process with the contents of a binary that is specified in a parameter
- The combination of these two, along with wait, allows you to build a shell
  - **wait** = a syscall that will return when a child process exits

---

2019-09-16

How do syscalls work?

- Syscalls have a one-to-one correspondence with actual routines in the kernel executable
  - *So, why not just call those methods from our program?*
    - Answer: isolation cannot be maintained with a simple routine call instruction

Isolation

- **Isolation** = the principle that processes that run in userland should not have the ability to interfere with each other's execution
  - This is achieved using a few techniques
    - **Address space** = the space of userland memory locations that a process can touch
    - **Privelege mode** = hardware support for conditionally enabling access to hardware based on a running program's privelege
    - **Well-written syscall code** = a kernel's implementation of the syscall interface is the main threat surface when isolation can be compromised

Tale of a syscall

- Issuing a syscall follows the same calling convention that normal procedure calls do
  - Arguments go into the registers a0-an
- `ecall` instruction does several things
  - Saves userspace PC into special register used exclusively for that purpose
  - Sets PC to an instruction address register that is populated by the kernel
    - That address points to the **trampoline**, the region of userspace memory that performs part of the transition to supervisor/kernel mode
      - The code in the trampoline backs up userspace execution state into another region of userspace memory caled the **trapframe**
  - Sets cpu to **supervisor mode**
    - **supervisor mode** = the level of hardware privelege that enables unrestricted access to hardware resources (*e.g.* memory)
- Once ecall moves execution to the trampoline, user execution state (registers, PC, *etc*) get backed up into the trapframe
- Once user execution state is saved to the trapframe, execution switches to `usertrap`, which is located in the kernel

  - This does some additional processing on the trapframe and then uncovers the reason why a usertrap was triggered (maybe a hardware exception, a syscall, or a timer interrupt) and dispatches to the correct piece of kernel handler code for that reason

---

2019-09-18

Shared Memory and Isolation

- The presence of multiple processes sharing one physical memory introduces difficulties enforcing isolation between processes
- Strategies for enforcing isolation
  - Construct programs in a way that ensures that memory boundaries are respected
    - Con: *What if someone writes a program that doesn't play nice with that?*
  - Use **virtual memory**
    - **Virtual memory** = a scheme for mapping virtual addresses to physical addresses using page tables
    - Con: some amount of hardwawre support is necessary for ensuring the translation is performed when it needs to happen
      - **MMU** = memory management unit; responsible for performing the translation

Benefits of virtual memory

- Because the resolution process only depends on state stored in physical memory, the kernel can directly manipulate that state to implement optimizations and features
  - *e.g.* COW, lazy allocation, *etc*

2019-09-23

Virtual Memory 2: Electric Boogaloo

- This lecture goes over some cool things that kernels can use virtual memory to implement interesting operating system features

Page Table Entries (PTE)

- **Page table entry** = an entry in a page table that stores information about a page's physical page numbers and metadata (permissions, access data, *etc.*)
  - These usually have empty space within so that OS's can place new metadata that will only have meaning to that specific OS

Page Faults

- **Page fault** = an event that is triggered by the MMU when a specific error relating to the translation of a virtual address into a physical address
  - How do we gather information about a page fault when it ocurrs?
    - **scause register (Supervisor CAUSE)** = a register an integer that stores the reason why a fault was triggered
    - **stval register** = the virtual address that caused the page fault
    - **User PC** = the program counter when the fault was triggered
    - **SSTATUS** = the state of whether the fault ocurred in supervisor mode or user mode
- In general, page faults are our most versatile tool for implementing features using virtual memory

Optimization: Lazy Allocation

- Oftentimes, a user program will ask for way more memory than it actually needs
  - **sbrk** = a syscall that asks the kernel to enlarge the heap region of memory
    - Technically, segmenting of a user address space can largely be managed by the user process; however, using the syscall gives you the convenience of checking for errors in growing the kernel
- We can simply return immediately and then allocate pages as the user program uses them
  - Called **lazy allocation**
  - The benefit is that in cases where a user program *doesn't* use all the memory they ask for, the overhead of allocating all that space isn't incurred
  - The downside is that in cases where a user program *does* use all the memory they ask for, the overhead of faulting for each new incremental allocation can be expensive

Optimization: Zero Pages

- Many situations call for sections of zeroed-out memory, often set to read-only
  - These often act as **guard pages**

        * **Guard page** = a page that exists after or before some region of memory that exists to cause a page fault if memory access strays beyond the confines of the memory location
            - *e.g.* the stack is surrounded by
- You can actually just allocate one physical page and alias every zero page to point to that page
    - This saves some amount of space that would otherwise be pointless to allocate for every
    - If some user program does need to write to the zero page (assuming it isn't intended as a guard page, you can use **copy on write (COW)** to duplicate the physical page so that isolation is maintained
        * **Copy on write (COW)** = an optimization technique where
            - Can also be used to optimize the fork syscall

Optimization: Demand Paging

- Modern operating systems often support **dynamic linking**
    - **Dynamic linking** = a system of making shared library code available to different processes through loading in that binary data at runtime
        * Oftentimes, shared object files are huge, and user programs only use small parts of the library
            - The kernel can page in parts of the shared object file as they're used

---

2019-09-25

Wherefore art thou, context switch?

- In general, times when a user program is interrupted are split into three categories
    1. Exceptions = *unintended* interruptions that happen because what is happening in the user program
    2. System calls = *intended* interruptions that happen because what is happening in the user program
    3. Interrupts = interruptions that happen because of some peripheral event that has nothing to do with what is happening in the user program
- **trap** = a term that sometimes refer to any one of of these three categories, but can also refer specifically to exceptions

Tale of a keyboard interrupt

- The kernel has all kinds of facilities for setting what kinds of interrupts it wants to handle and when

- **driver** = some piece of kernel code that manages interactions with an I/O device
  - Usually split into two conceptual parts
    - **top** = the part of the driver that implements some syscall interface for that device
    - **bottom** = the part of the driver that handles interrupts from that device
    - Since the interrupts happen asynchrounously, some kind of synchronization scheme is necessary within the two halves of a driver
- Hardware for supporting interrupts
  - **sstatus** = supervisor status register
  - **sie** = supervisor interrupt enable register
  - **sip** = interrupt pending register
  - **scause** = stores the information about the cause for the interrupt
  - **stvec** = stores the program counter to return to after handling the interrupt

---

2019-09-30

Multicore parallelism

- Moore's law is beginning to break down
  - Modern strategies to accelerate workloads involve parallelism in some form or another
- Modern OS's supply **synchronization primitives**
  - **Synchronization primitive** = a syscall that alllows coordination of access to program state or timing between user threads or between processes
    - *e.g.* semaphores, mutex's, barriers, *etc.*

Lock granularity

- **Coarse-grain locking** = a strategy for synchronization which involves having a small set of locks that regulate a lot of concurrent code paths at once
  - Pros: easy to write, harder to disobey locking discipline
  - Cons: many threads competing for one lock leads to poor throughput
- **Fine-grain locking** = a strategy for synchronization which involves having a large set of locks that aim to regulate a smaller set of competing threads
  - Pros: faster
  - Cons: harder to write
  - *So, fine-grain locking is always the best strategy?*

        * Well, no. Sometimes, the overhead of parallelism is significant enough in comparison to the benefits that a single threaded approach is best
            · It really depends on the specifics of the workload and the hardware that you're running on

How do we write properly synchronized code?

1. Write code that is semantically correct if serial execution is guaranteed
2. Guarantee serial execution by inserting synchronization primitives
    - In general, it's wise to start with coarser-grain locks and *then* move towards finer grain locks if you observe that execution is bottlenecking in a way that limits overall throughput of your system
        – `Premature optimization is the greatest evil`

How do operating systems implement locks

- Naive approach doesn't work, because C doesn't have a facility for making sure
- Copilers and CPUs can actually reorder
    – **Memory model** = a set of legal modifications to a program's access pattern of memory
        * RISC-V has a very relaxed memory model, which permits all kinds of reordering
- In order to sidestep thene memory reorders, instruction sets usually provide **atomic instructions**
    – **Atomic instructions** = an instruction in an ISA can read from memory, perform an operation, and then write to memory, all atomically
        * *e.g.* SWAP
    – **Fence** = a RISC-V instruction that imposes constraints on the reordering of memory accesses