Introduction to xv6

Mythili Vutukuru

CSE, IIT Bombay

xv6: a simple teaching OS

- xv6 is a simple OS for easy teaching of OS concepts
  - Two versions, one for x86 hardware and one for RISC-V hardware
  - This series of lectures based on x86 version
  - https://github.com/mit-pdos/xv6-public
- Easy to read code, simple exercises to write OS code
- Much simpler than real OS like Linux, but basic concepts remain the same
- Runs inside QEMU emulator

OS code in C or assembly?

- OS is also like any other program run by CPU, but it is the most important program that manages other programs
  - OS code mostly written in a high-level language like C, compiled into executable, loaded at boot time
- But some parts of OS are written directly in assembly language or CPU instructions that the hardware can understand?
  - Why not write everything in C? Not possible to express certain low level actions performed by OS in high level language
- Basic understanding of x86 assembly code required for understanding xv6 OS code in this course

Reference: x86 registers

- General purpose registers: store data during computations (eax, ebx, ecx, edx, esi, edi)
- Pointers to stack locations: base of stack (ebp) and top of stack (esp)
- Program counter or instruction pointer (eip): next instruction to execute
- Control registers: hold control information or metadata of a process (e.g., cr3 has information related to memory of process)
- Segment registers (cs, ds, es, fs, gs, ss): information about segments (related to memory of process)

Reference: x86 instructions

- Load/store: *mov src, dst*
  - *mov %eax, %ebx* (copy contents of eax to ebx)
  - *mov (%eax), %ebx* (copy contents at the address in eax into ebx)
  - *mov 4(%eax), %ebx* (copy contents stored at offset of 4 bytes from address stored at eax into ebx)
- Push/pop on stack: changes esp
  - *push %eax* (push contents of eax onto stack, update esp)
  - *pop %eax* (pop top of stack onto eax, update esp)
- *jmp* sets eip to specified address
- *call* to invoke a function, *ret* to return from a function
- Variants of above (*movw, pushl*) for different register sizes

Reference: Mechanics of function call

- Local variables, arguments, return address stored on stack for duration of function call
- What happens in a function call?
  - Push function arguments on stack
  - *call fn* (instruction pushes return address on stack, jumps to function)
  - Allocate local variables on stack
  - Run function code
  - *ret* (instruction pops return address, eip goes back to old value)
- All of this is automatically done by the C compiler for you, and is part of the C calling convention.

Reference: Caller and callee save registers

- What about values in registers that existed before function call? Registers can get clobbered during a function call, so how can computation resume?
  - Some registers saved on stack by caller before invoking the function (caller save registers). Function code (callee) can freely change them, caller restores them later on.
  - Some registers saved by callee function and restored after function ends (callee save registers). Caller expects them to have same value on return.
  - Return value stored in eax register by callee (one of caller save registers)
- All of this is automatically done by C compiler (C calling convention)

Reference: More details of function call

- Anatomy of a function call
  - Push caller save registers (eax, ecx, edx)
  - Push arguments in reverse order
  - Return address (old eip) pushed on stack by the call instruction
  - Push old ebp on stack
  - Set ebp to current top of stack (base of new “stack frame” of the function)
  - Push local variables and callee save registers (ebx, esi, edi)
  - Execute function code
  - Pop stack frame and restore old ebp
  - Return address popped and eip restored by the ret instruction
- Stack pointers: ebp stores address of base of current stack frame and esp stores address of current top of stack
  - Function arguments are accessible from looking under the stack base pointer