Lecture 21: xv6 introduction and x86 background

Mythili Vutukuru

IIT Bombay

https://www.cse.iitb.ac.in/~mythili/os/

Understanding OS concepts using xv6

- Deep dive into OS concepts using xv6 as example
  - See complete lecture series at https://www.cse.iitb.ac.in/~mythili/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
- This lecture: overview of x86 hardware and other background needed to understand xv6 code

Understanding xv6: background

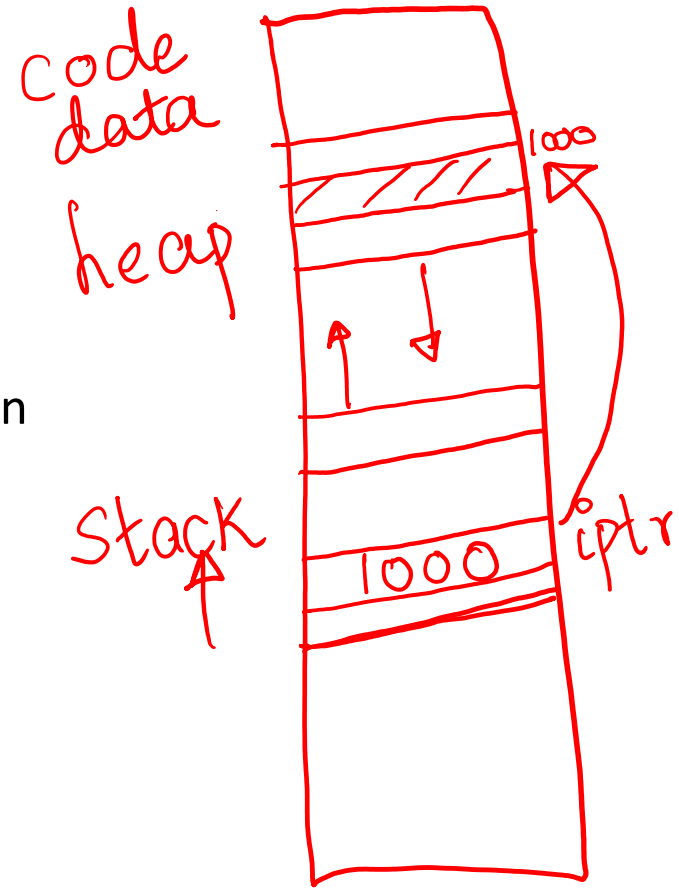
- OS enables processes stored in memory to run on CPU
  - Process code/data in main memory
  - CPU fetches, decodes, executes instructions in program code
  - Process data fetched from memory to CPU registers for faster access during instruction execution
  - Recently fetched code/data stored in CPU caches for future access (memory access is very slow compared to CPU)
- What we will cover in this lecture
  - Common x86 registers
  - Common x86 instructions
  - How stack is used during function calls (C calling convention)

Memory image of a process

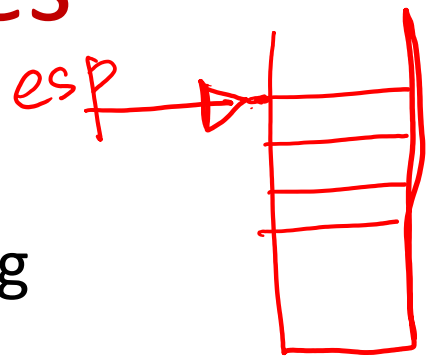
- Process memory image consists of
  - Compiled code (CPU instructions)
  - Global/static variables (memory allocated at compile time)
  - Heap (dynamic memory allocation via, e.g., malloc) that grows on demand
  - Stack (temporary storage during function calls, e.g., local variables) that usually grows "up" towards lower addresses
  - Other things like shared libraries
- Every instruction/data has an address, used by CPU to fetch/store
  - Virtual addresses (managed by OS)
- Example: can you understand what is happening with variable "iptr"?

It is getting a virtual address.

```
int *iptr = malloc(sizeof(int))
```



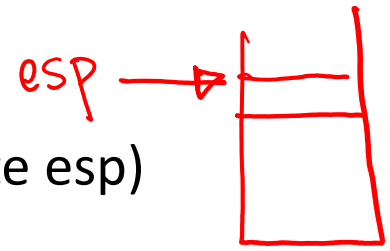
x86 registers: examples



- Small space for data storage within CPU
- 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 pointer to page table)
- Segment registers (cs, ds, es, fs, gs, ss): information about segments (related to memory of process)

x86 instructions: examples

- Load/store: *mov src, dst* (AT&T syntax)
  - *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

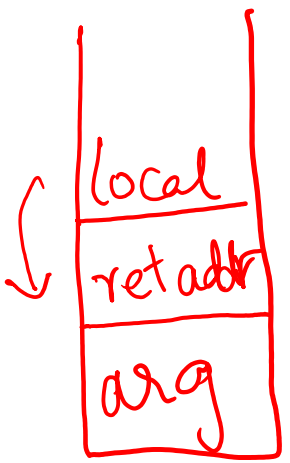


Privilege levels

- x86 CPUs have multiple privilege levels
  - Four "rings" (0 to 3)
  - Ring 0 has highest privilege, runs OS code
  - Ring 3 has lowest privilege, runs user code
- Two types of instructions: privileged and unprivileged
- Privileged instructions can be executed by CPU only when running at the highest privilege level (ring 0)
  - For example, writing into cr3 register (setting page table) is privileged instruction, only OS should do it, because we do not want a user manipulating memory of another process
  - Another example: instructions to access I/O devices
- Unprivileged instructions can be run at lower privilege levels
  - For example, user code running at lower privilege can store a value into a general purpose register
- When user requires OS services (e.g., system call), CPU moves to higher privilege level and executes OS code that contains privileged instructions
  - User code cannot invoke privileged instructions directly

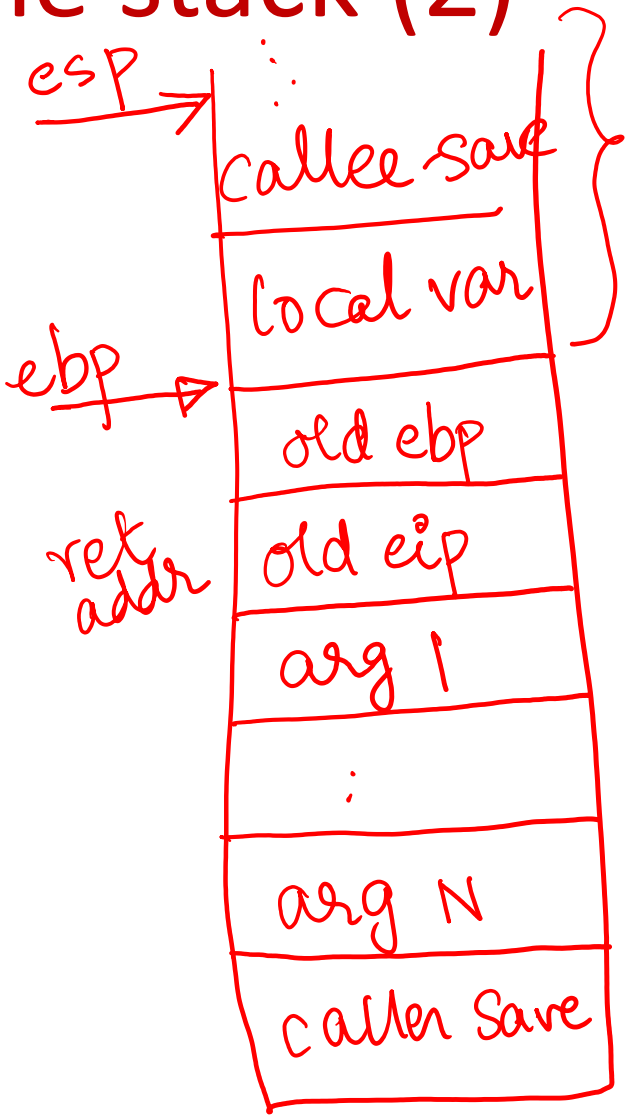
Function calls and the stack (1)

- Local variables, arguments 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)
- 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)



Function calls and the stack (2)

- Timeline of a function call (note: stack grows from "up" from higher to lower addresses)
  - 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



C vs. assembly for OS code

- Why all this x86 background?
  - Most of xv6 is in C, and assembly code (including all the stack manipulations for function calls) is automatically generated by compiler.
  - However, small parts are in assembly language. Why? Sometimes, OS needs more control over what needs to be done (for example, the logic of switching from stack of one process to stack of another cannot be written in a high-level language)
  - Basic understanding of x86 assembly language is required to follow some nuances of xv6 code

More on CPU hardware

- Some aspects of CPU hardware that are not relevant to studying OS:
  - CPU cache: CPU stores recently fetched instructions and data in multiple levels of cache. OS has no visibility or control into the CPU cache.
  - Hyper-threading: A CPU core can run multiple processes concurrently via hyper-threading. From an OS perspective, 4-core CPU with 2 hyper-threads per core, and 8-core CPU with no hyper-threading will look the same, even though performance may differ. OS will schedule processes in parallel on the 8 available processors.

Using xv6 code

- git clone https://github.com/mit-pdos/xv6-public
- Inside the code directory, do “make” to compile
- xv6 runs on a hardware emulator called QEMU
  - Do “make qemu” or “make qemu-nox” to run xv6
  - Can also connect gdb to QEMU for debugging
- After bootup, xv6 opens a shell in which common commands and other user programs can be run
- Reading xv6 code
  - Read through the source code files
  - Use PDF formatted source code (easier to reference with page numbers and line numbers)
  - More info at https://www.cse.iitb.ac.in/~mythili/os/