System calls, C, inline assembler

Michal Sojka
sojkam1@fel.cvut.cz
ČVUT, FEL

License: CC-BY-SA 4.0
Review: Hello world in assembler

```
hello:
    .ascii "Hello world\n"

.global _start
_start:
    mov $4,%eax     # write
    mov $1,%ebx     # stdout
    mov $hello,%ecx # ptr to data
    mov $12,%edx    # length of the data
    int $0x80
```

- Compile: `gcc -m32 -nostdlib -o hello1 hello1.S`
- Run: `./hello1`
Hello world in C with inline assembler

char *hello = "Hello world\n";

void _start()
{
    asm volatile ("mov $4,%eax;"
                  "mov $1,%ebx;"
                  "mov hello,%ecx;"
                  "mov $12,%edx;"
                  "int $0x80"
                  );
}

- But C compiler allows us to do better than that!
  - Assembler instructions with C expressions as operands

Compilation:
gcc -m32 -nostdlib -nostdinc -static -O2 hello2.c -o hello2
Extended assembler

// Compile with gcc -m32 -O2 -Wall ...
#include <stdio.h>
int main()
{
    void *stack_ptr;
    asm volatile (
        "mov %%esp,%0;" : "=g" (stack_ptr));
    printf("Value of ESP register is %p\n", stack_ptr);
    return 0;
}

• Allows using C expressions in assembler instructions
• Programmer writes “instruction templates”
• Compiler replaces parameters (%0 above) with real operands (registers, memory references, ...)
• Compiler does not try to understand the asm code! Programmer has to tell what is the effect of the assembler.
```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int result, op1 = 4, op2 = 2;
    asm volatile (    
        "mov %1,%0;" 
        "add %2,%0;" 
        : "=r" (result) 
        : "r" (op1), "r" (op2) 
        : "cc"; // flags register (condition codes) is modified
    printf("result = %d\n", result);
    return 0;
}
```

Compiles into (objdump -d ...):
```
80482c0:       ba 02 00 00 00          mov    $0x2,%edx
80482c5:       b8 04 00 00 00          mov    $0x4,%eax
80482ca:       89 c0                         mov    %eax,%eax
80482cc:       01 d0                         add     %edx,%eax
...```
Extended assembler constraints

- Tell the compiler which registers or other operands are allowed in instructions given in the template
  - Generic constraints
    - “g” – anything
    - “r” – register:
      \[ \text{asm volatile ("mov %0,%%eax" :: "r" (var) : "eax")} \rightarrow \text{mov %ebx,%%eax} \]
    - “m” – memory:
      \[ \text{asm volatile ("mov %0,%%eax" :: "m" (var) : "eax")} \rightarrow \text{mov var,%%eax} \]
    - “i” – immediate operand:
      \[ \text{asm volatile ("mov %0,%%eax" :: "i" (123) : "eax")} \rightarrow \text{mov $123,%%eax} \]
  - Machine (HW) specific constraints
    - “a” – *ax register (for x86)
    - “b” – *bx register (for x86)
    - ...
Hello world in C with extended assembler

void _start()
{
    char hello[] = "Hello world\n";
    int retval;
    asm volatile ("int $0x80"
                  : "=a" (retval)
                  : "a" (4), "b" (1), "c" (hello), "d" (sizeof(hello)-1)
                  : "memory");
    asm volatile ("int $0x80" : : "a" (1), "b" (0));
}

- “memory” in clobber list, tells the compiler that the syscall touches memory and the content of the hello variable cannot be optimized out (try removing it)
- Compile: gcc -m32 -nostdlib -nostdinc -static -Wall -O2 hello3.c \
  -o hello3
- Disassemble: objdump -d hello3
Review: Linux system calls (x86, 32-bit)

- Application Binary Interface
  - int 0x80 (older, simpler, slower)
    - System call number in EAX
      - `/usr/include/sys/syscall.h`
      - `/usr/include/asm/unistd_32.h`
      - Note: Different architectures (e.g. x86_64) use different system call numbers.
  - Arguments
    - 1\(^{st}\) in EBX, 2\(^{nd}\) in ECX, 3\(^{rd}\) in EDX, 4\(^{th}\) in ESI, 5\(^{th}\) in EDI, 6\(^{th}\) in EBP
    - More arguments need to be passed in memory pointed at by a register
  - Return value: EAX
    - Zero or positive: success
    - Negative: error (see `/usr/include/asm-generic/errno.h`, `errno-base`)
    - `sysenter` (newer, faster, slightly more complicated)

- Documentation (arguments)
  - `man 2 syscall_name`
  - `man 2 write`
C wrappers around system calls

static inline long syscall1(long syscall, long arg1) {
    long ret;
    asm volatile ("int $0x80" : "=a" (ret) : "a" (syscall), "b" (arg1) : "memory")
    return ret;
}

static inline long syscall3(long syscall, long arg1, long arg2, long arg3) {
    long ret;
    asm volatile ("int $0x80" : "=a" (ret) : "a" (syscall), "b" (arg1), "c" (arg2),
                 "d" (arg3) : "memory")
    return ret;
}

int write(int fd, const void *buf, int count) {
    return syscall3(4, fd, (long)buf, count);
}

void exit(int status) {
    syscall1(1, status);
}

void _start() {
    int retval;
    retval = write(1, "Hello world\n", 12);
    exit(0);
}
Assignment

• Write a program that:
  – Opens file “file.txt” \texttt{(open())}
  – Reads the first 100 bytes of the file \texttt{(read())}
  – Writes the first line (or 100 bytes if the line is longer) of the read data to standard output \texttt{(write())}
  – Executes program /bin/date \texttt{(execve())}

• The program must compile for i386 \texttt{without libc}
i.e. with \texttt{gcc -m32 -nostdlib -nostdinc} ...