System calls and assembler

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

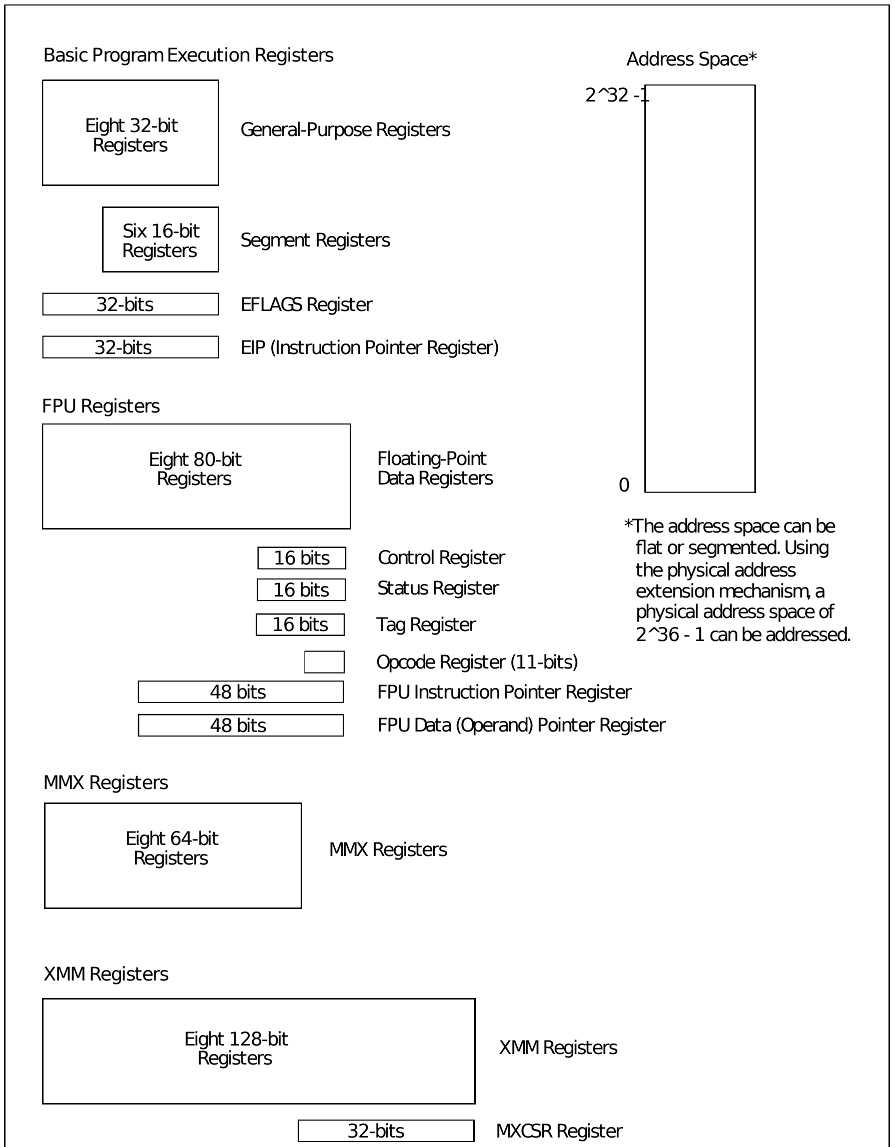
System calls
(repetition from lectures)

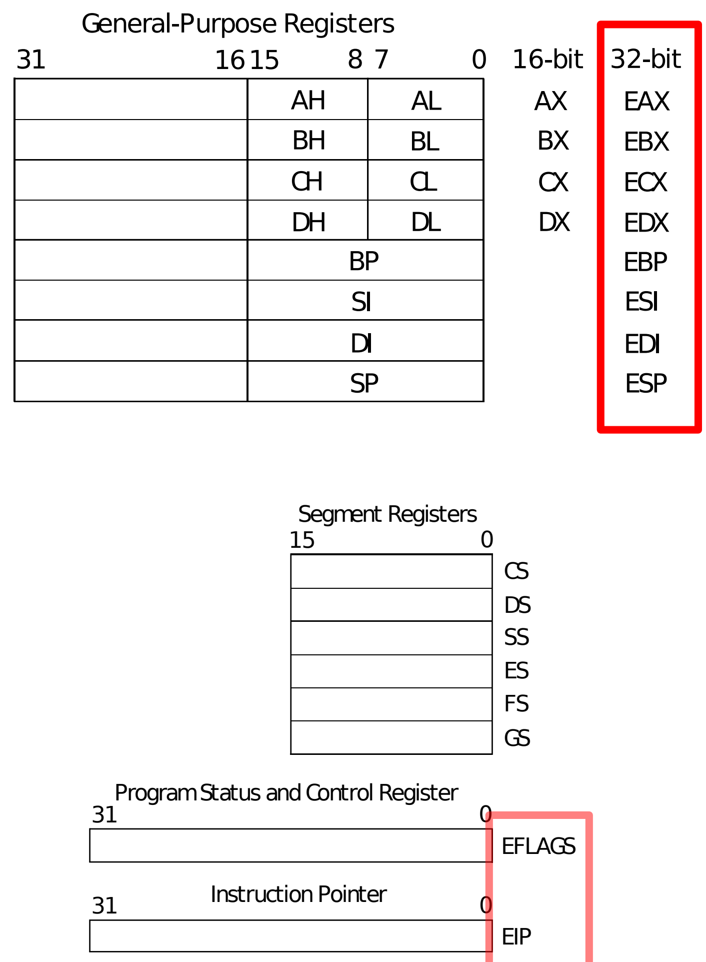
- A way for “normal” applications to invoke operating system (OS) kernel's services.
- Applications run in unprivileged CPU mode (user space, user mode)
- OS kernel runs in privileged CPU mode (kernel mode)
- System call is a way to securely switch from user to kernel mode.

What is a system call technically?

- A machine instruction that:
  - Increases the CPU privilege level and
  - Passes the control to a predefined place in the kernel.
- Arguments are (typically) passed in CPU registers.
- Instructions:
  - x86: int 0x80, sysenter, syscall
  - MIPS: syscall
  - ARM: swi

x86 user execution environment
(32 bit)





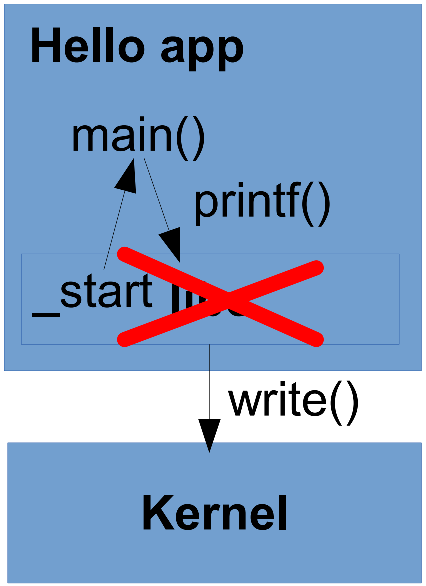
Source: Intel

Linux system call ABI
(x86, 32-bit)

- Application Binary Interface
- Two different interfaces:
  - 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
      - 1st in EBX, 2nd in ECX, 3rd in EDX, 4th in ESI, 5th in EDI, 6th 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)

Hello world

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  printf(“Hello world\n”);
}
```



- Let's look how to do it without libc
- write system call
  - Documentation: `man 2 write`
  - `ssize_t write(int fd, const void *buf, size_t count);`
  - Three arguments
- _startup symbol

Hello world (taken from lectures)

It's simpler in assembler

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

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

AT&T assembler syntax:

label:

    instruction src,dst

.directive

- immediate operands preceded by '$'
- register operands preceded by '%'
- label/number without '$' or '%' means reading or writing from/to the given memory address!

https://sourceware.org/binutils/docs/as/Syntax.html
https://sourceware.org/binutils/docs/as/i386_002dSyntax.html

- Compile: `gcc -m32 -nostdlib -o hello1 hello1.s`
- Run: `./hello1`
- Why it ends with segmentation fault?
- Disassemble the binary: `objdump -d hello1`

Getting rid of the fault

```
hello:                    # initialized va
        .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 
        int $0x80

        mov $1,%eax       # exit
        mov $0,%ebx       # exit code
        int $0x80
```

- We need to tell the OS that we are about to finish – with `exit` syscall.
- Inspect the syscalls invoked by the program:

```
strace ./hello > /dev/null
```

Assignment

- Write an assembler equivalent of the program from the next slide.
- **Input:** One digit 0 – 9 read from stdin.
- **Output:** Two digit decimal number on stdout – it is the Fibonacci number corresponding to the input.
- You don’t need to check for run-time errors.
- The resulting “stripped” binary should be smaller than 1 KiB.

fibo.c

```
#include <unistd.h>

int fibo(int n)
{
	if (n < 2)
		return 1;
	else
		return fibo(n - 2) + fibo(n - 1);
}
```

```
int main(int argc, char *argv[])
{
	char n, str[3];
	int val;

	read(0, &n, sizeof(n));

	n = n - '0';
	val = fibo(n);

	str[0] = '0' + (val / 10);
	str[1] = '0' + (val % 10);
	str[2] = '\n';

	write(1, str, sizeof(str));

	return 0;
}
```

Useful instructions

- mov – moves data between registers and memory
  - `mov $1,%eax   # move 1 to register eax`
  - ```
    n: .int 123   # label n points to an integer
                # variable
    mov n,%eax    # move value of the variable to eax
    ```
  - `mov %eax,%ebx # copy the value in eax to ebx`
- push/pop – stack manipulation
  - Useful when we need to store data for later and we cannot use registers for that
  - `push %eax    # push content of eax to the stack`
  - `pop %ebx     # pop a value from the stack to ebx`
    ```

Useful instructions 2

- add – adds two operands
  - add $2,%eax   # eax = eax + 2
  - add %eax,%ebx # ebx = ebx + eax
- sub – subtracts two operands
  - sub $2,%eax   # eax = eax – 2

Useful instructions 3

- call – calls a subroutine
- ret – returns from a subroutine to the caller

```
plusone:
    add $1,%eax
    ret
main:
     mov $12,%eax
     call plusone
     ...
```

Useful instructions 4

- div – integer division (not a simple instruction)
  http://x86.renejeschke.de/html/file_module_x86_id_72.html
  - 8 bit operand: ax divided by the operand
    result: al = ax / operand, ah = ax % operand
    - ```
      mov $42,%ax
      mov $12,%bl
      div %bl      # al = 42/12 = 3
      ```
  - 16 bit operand: dx:ax divided by the operand
    result: ax = dx:ax / operand, dx = dx:ax % operand
    - ```
      mov $0x1,%dx
      mov $0x2345,%ax
      mov $10,%bx
      div %bx      # ax = 0x12345 / 10
      ```
  - ...

Useful instructions 5

- cmp – compare two values
  - cmp $2,%eax # compare eax with 2 and set eflags register
  - je label   # jump to the label if eax was **equal** to 2
  - jl label   # jump if eax was **less**
  - jg label   # jump if eax was **greater**
  - jlelabel   # jump if **less or equal**
  - jge label   # jump if **greater or equal**
- Example**:**
  - ```
        cmp $0x30,%al
        jl nodigit
        cmp %0x39,%al
        jg nodigit
    digit:
        ... do something ...
    nodigit:
        ... handle error
    ```
    ```

Makefile

```
all: myfibo

%: %.s    # Disable built-in rule

%.o: %.s    # Assembler rule (produce 32 bit code even on 64 bit system)
	as --32 -g -o $@ $<

%: %.o    # Linker rule
	ld -m elf_i386 -g -o $@ $<
```

Troubleshooting

- $ ./myfibo
  9080 segmentation fault ./myfibo
  $ gdb myfibo
  (gdb) run
  Program received signal SIGSEGV, Segmentation fault.
  _start () at myfibo.s:30
  30   mov   3,%eax
- Fix: mov $3,%eax

Other hints

- man ascii
- info as
- gdb: info reg
- https://en.wikipedia.org/wiki/X86_instruction_listings
- http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
- Write your code in C and use “objdump -d” to look at the assembler instructions generated by the compiler.