CS162
Operating Systems and Systems Programming
Lecture 3

Processes (Continued)

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

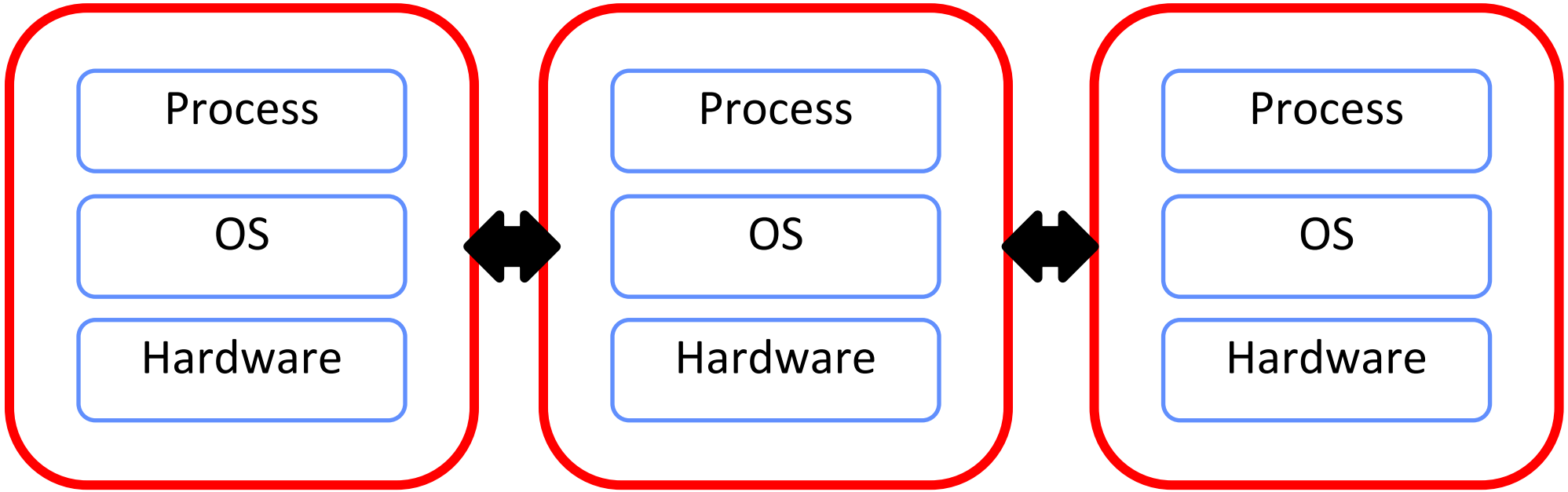
Reminder

Drop now (by Jan 31st) or forever hold your peace

(aka stay enrolled in CS162)

Recall: The Process

A executing program with restricted rights



Enforcing mechanism must not hinder functionality or hurt performance

Recall: Operating System Kernel

Lowest level of OS running on system.

Kernel is trusted with full access to all hardware capabilities

All other software (OS or applications) is considered untrusted

Untrusted	Applications
	Rest of OS
Trusted	Operating System Kernel
Untrusted	Hardware

Recall: Dual Mode Operation

Use a bit to enable two modes of execution

In User Mode	In Kernel Mode
Processor checks each instruction before executing it	OS executes with protection checks off
Executes a limited (safe) set of instructions	Can execute any instructions

Recall: Hardware must support

1) Privileged Instructions

Unsafe instructions cannot be executed in user mode

2) Memory Isolation

Memory accesses outside a process's address space prohibited

3) Interrupts

Ensure kernel can regain control from running process

4) Safe Transfers

Correctly transfer control from user-mode to kernel-mode and back

Goals for today

- What hardware support is necessary to enable protection?
- 61C Review: The Stack?
- How to switch from user mode to kernel mode and back?

Hardware must support

1) Privileged Instructions

Unsafe instructions cannot be executed in user mode

2) Memory Isolation

Memory accesses outside a process's address space prohibited

3) Interrupts

Ensure kernel can regain control from running process

4) Safe Transfers

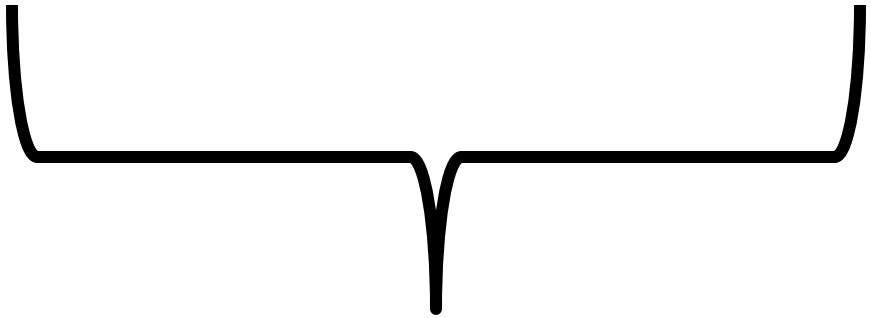
Correctly transfer control from user-mode to kernel-mode and back

Req 4/4: Safe Control Transfer

How do safely/correctly transition from executing user process to executing the kernel?

1) System Calls

2) Exceptions



Synchronous Events (trapping)

3) Interrupts

Asynchronous

Can be maskable or non-maskable

Safe Control Transfer: System Calls

User program requests OS service
Transfers to kernel at well-defined location

Synchronous/non-maskable

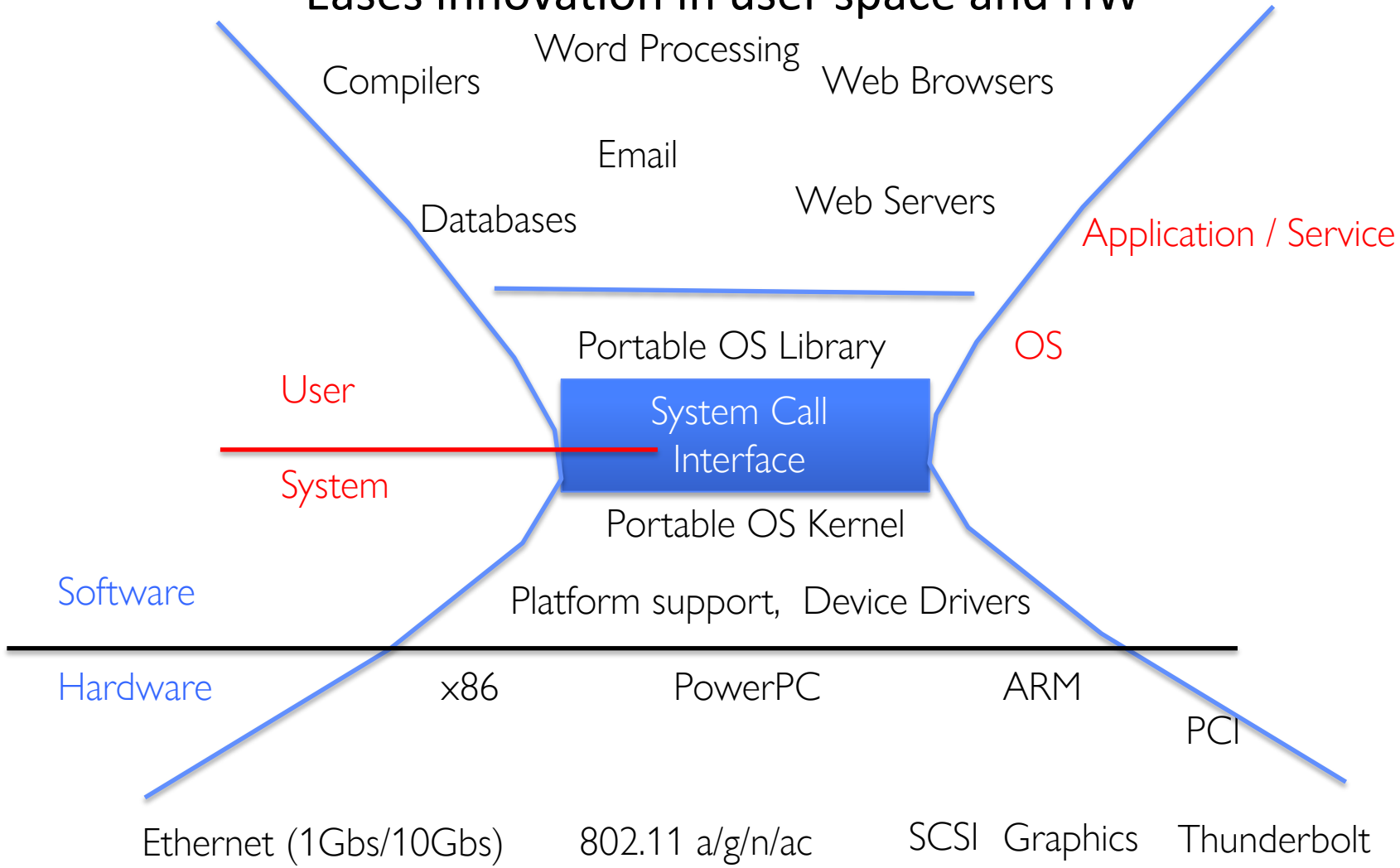
Read input/write to screen, to files, create new processes, send network packets, get time, etc.

How many system calls in Linux 3.0 ?
a) 15 b) 336 c) 1021 d) 21121

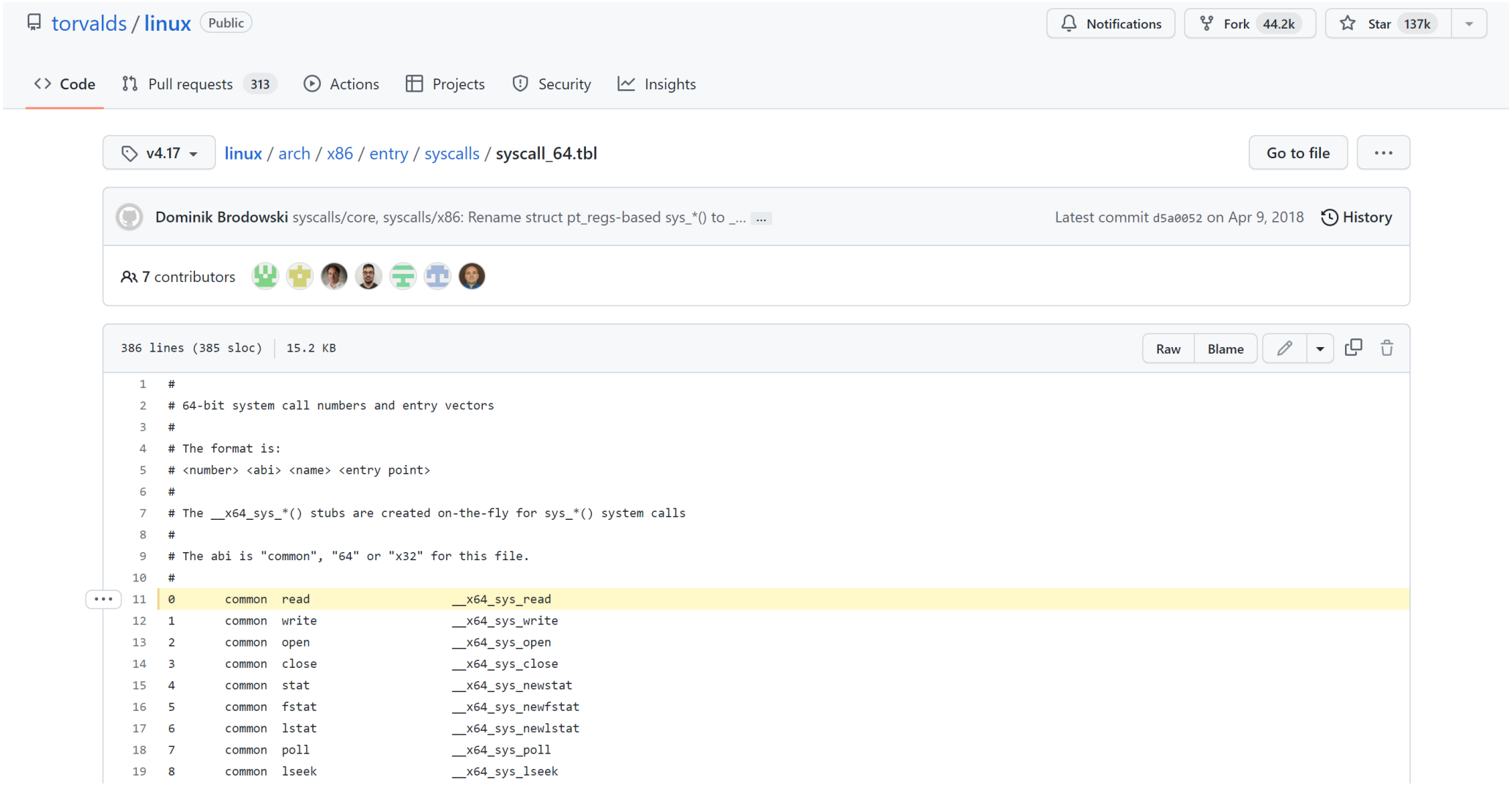
https://man7.org/linux/man-pages/man2/syscalls.2.html

System Calls are the "Narrow Waist"

Simple and powerful interface allows separation of concern
Eases innovation in user space and HW



System Calls in the Wild (In Linux)



torvalds / linux Public

Notifications | Fork 44.2k | Star 137k

Code | Pull requests 313 | Actions | Projects | Security | Insights

v4.17 | linux / arch / x86 / entry / syscalls / syscall_64.tbl | Go to file | ...

Dominik Brodowski syscalls/core, syscalls/x86: Rename struct pt_regs-based sys_*() to _... ... Latest commit d5a0052 on Apr 9, 2018 | History

7 contributors

386 lines (385 sloc) | 15.2 KB | Raw | Blame

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    __x64_sys_read
1       common  write                   __x64_sys_write
2       common  open                    __x64_sys_open
3       common  close                   __x64_sys_close
4       common  stat                    __x64_sys_newstat
5       common  fstat                   __x64_sys_newfstat
6       common  lstat                   __x64_sys_newlstat
7       common  poll                    __x64_sys_poll
8       common  lseek                   __x64_sys_lseek
```

Safe Control Transfer: Exceptions

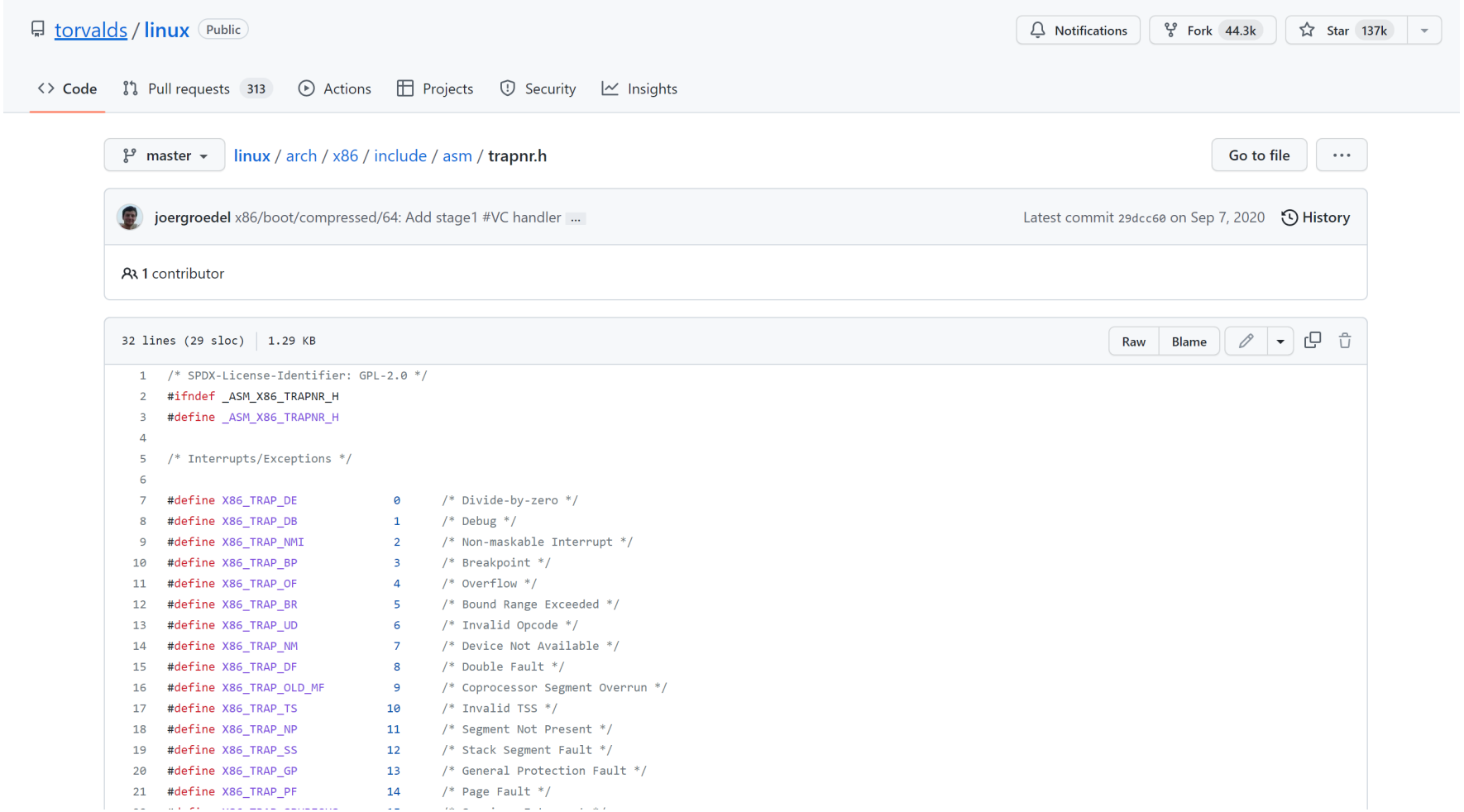
Any unexpected condition caused by user program behaviour

Stop executing process and enter kernel at specific exception handler

Synchronous and non-maskable

Process missteps (division by zero, writing read-only memory)
Attempts to execute a privileged instruction in user mode
Debugger breakpoints!

Exceptions in the Wild (In Linux)



torvalds / linux Public

Notifications | Fork 44.3k | Star 137k

Code | Pull requests 313 | Actions | Projects | Security | Insights

master | linux / arch / x86 / include / asm / trapnr.h | Go to file | ...

joergroedel x86/boot/compressed/64: Add stage1 #VC handler ... Latest commit 29dcc60 on Sep 7, 2020 History

1 contributor

32 lines (29 sloc) | 1.29 KB | Raw | Blame

```
1  /* SPDX-License-Identifier: GPL-2.0 */
2  #ifndef _ASM_X86_TRAPNR_H
3  #define _ASM_X86_TRAPNR_H
4
5  /* Interrupts/Exceptions */
6
7  #define X86_TRAP_DE             0      /* Divide-by-zero */
8  #define X86_TRAP_DB             1      /* Debug */
9  #define X86_TRAP_NMI            2      /* Non-maskable Interrupt */
10 #define X86_TRAP_BP             3      /* Breakpoint */
11 #define X86_TRAP_OF             4      /* Overflow */
12 #define X86_TRAP_BR             5      /* Bound Range Exceeded */
13 #define X86_TRAP_UD             6      /* Invalid Opcode */
14 #define X86_TRAP_NM             7      /* Device Not Available */
15 #define X86_TRAP_DF             8      /* Double Fault */
16 #define X86_TRAP_OLD_MF         9      /* Coprocessor Segment Overrun */
17 #define X86_TRAP_TS            10      /* Invalid TSS */
18 #define X86_TRAP_NP            11      /* Segment Not Present */
19 #define X86_TRAP_SS            12      /* Stack Segment Fault */
20 #define X86_TRAP_GP            13      /* General Protection Fault */
21 #define X86_TRAP_PF            14      /* Page Fault */
```

Safe Control Transfer: Interrupts

Asynchronous signal to the processor that some external event has occurred and may require attention

When process interrupt, stop current process and enter kernel at designated interrupt handler

Timer Interrupts, IO Interrupts, Interprocessor Interrupts

Safe Control Transfer: Kernel->User

New Process Creation

Kernel instantiates datastructures, sets registers, switches to user mode

Resume after an exception/interrupt/syscall

Resume execution by restoring PC, registers, and unsetting mode

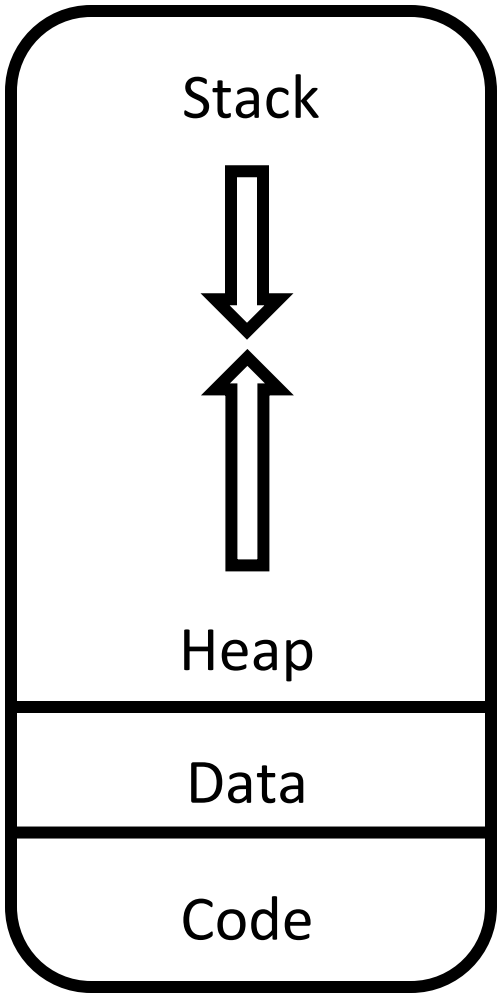
Switching to a different process

Save old process state. Load new process state (restore PC, registers). Unset mode.

Goal 2: The Stack is Back (Review)

Stack Contains temporary data such as method/function parameters, return address and local variables.

Heap Dynamically allocated memory to a process during its run time.




```
int foo() {
   int a;
   Foo* foo=  malloc(sizeof(int));

}
```

Stack Terminology (Review)

Stack Frame

All the information on the stack pertaining to a function call

Frame Pointer (%ebp)

Contain base address of function's frame.

Stack Pointer (%esp)

Points to the next item on the stack.

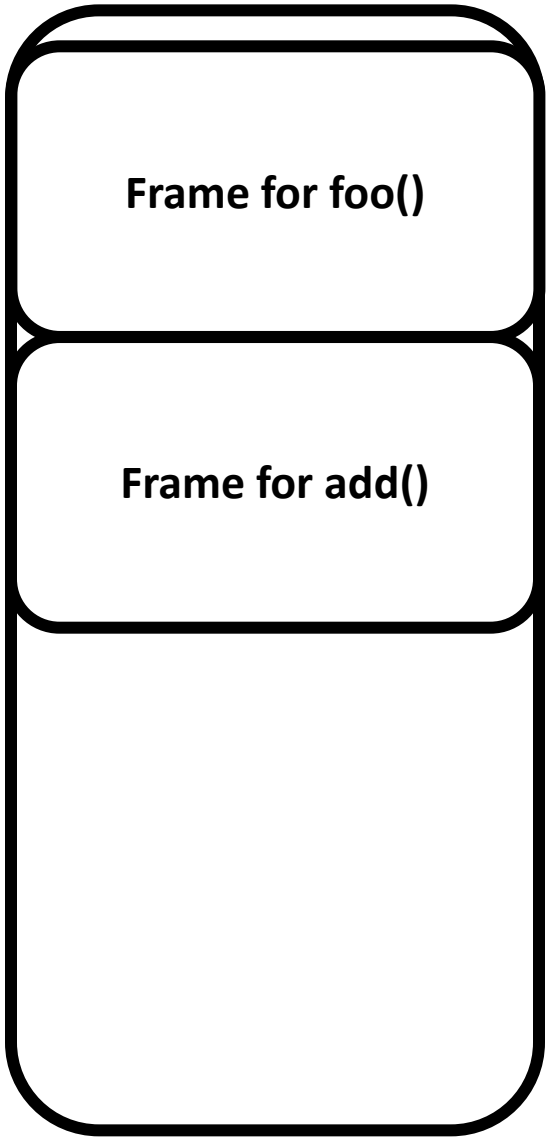
Instruction Pointer (%eip)

Indicates the current address of the program being executed

The Call Stack (Review)

```
int add(int a, int b) {
  int result = a+b;
  return result;
}


void foo() {
  int x = add(5,10);
}
```



0xFFFFFFFF

0x00000000

The Call Stack (Review)

```
int add(int a, int b) {
  int result = a+b;
  return result;
}


void foo() {
  int x = add(5,10);
}
```

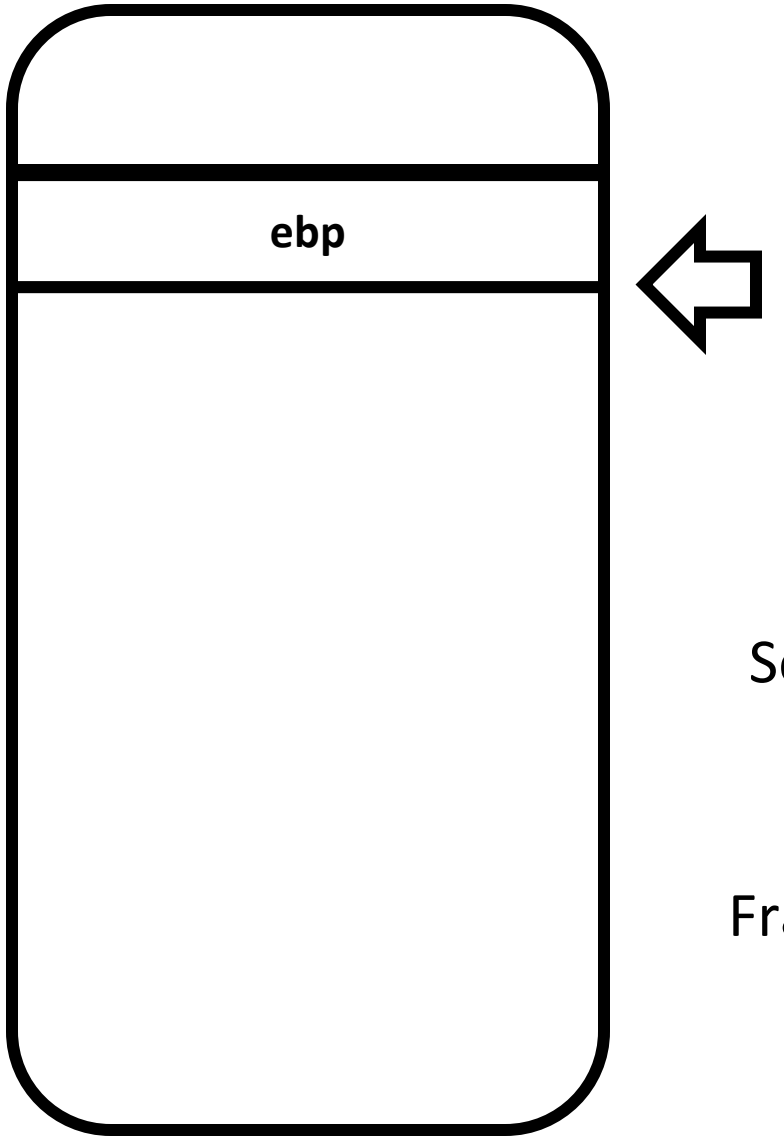
```
crooks@laptop> gcc -S  -m32 add.c
```

```
add:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave/ret
foo:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $10
        pushl   $5
        call    add
        movl    %eax, -4(%ebp)
        leave/ret
```

The Call Stack (Review)

```
void foo() {
  int x = add(5,10);
}
```

```
foo:
   pushl   %ebp
   movl    %esp, %ebp
```



Stack Pointer (esp)

Save old frame pointer.

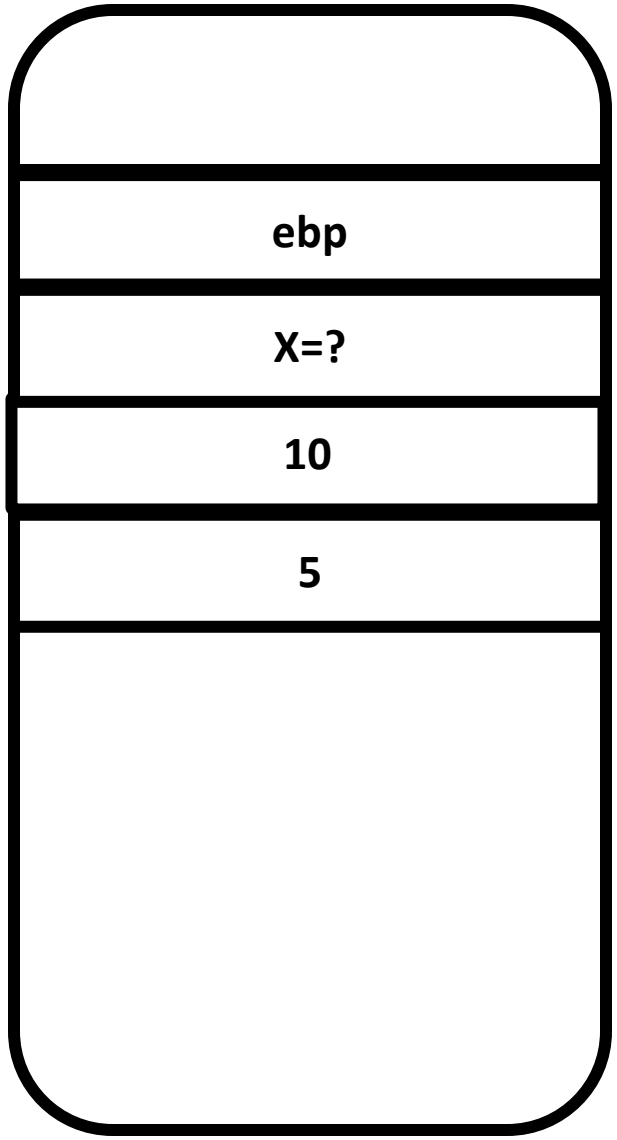
Set current frame pointer to stack pointer

Frame pointer is base of stack frame

The Call Stack (Review)

```
void foo() {
  int x = add(5,10);
}
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    pushl   $10
    pushl   $5
```



Stack Pointer (esp)

Stack Pointer (esp)

Stack Pointer (esp)

Create space for x

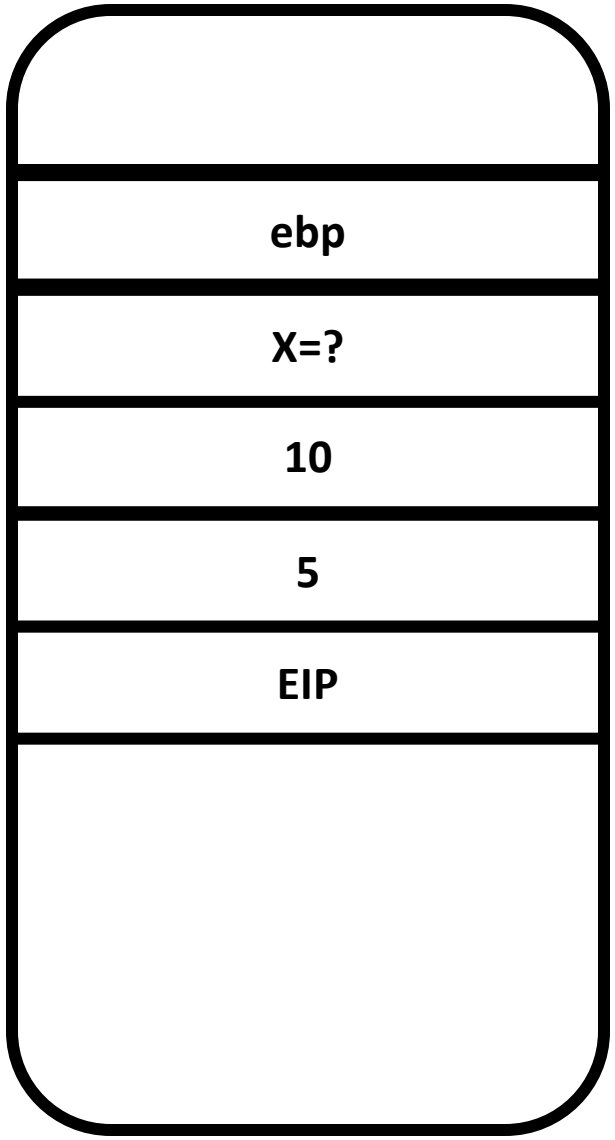
Load Function Parameters On Stack

(reverse order)

The Call Stack (Review)

```
void foo() {
  int x = add(5,10);
}
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    pushl   $10
    pushl   $5
    call add
```

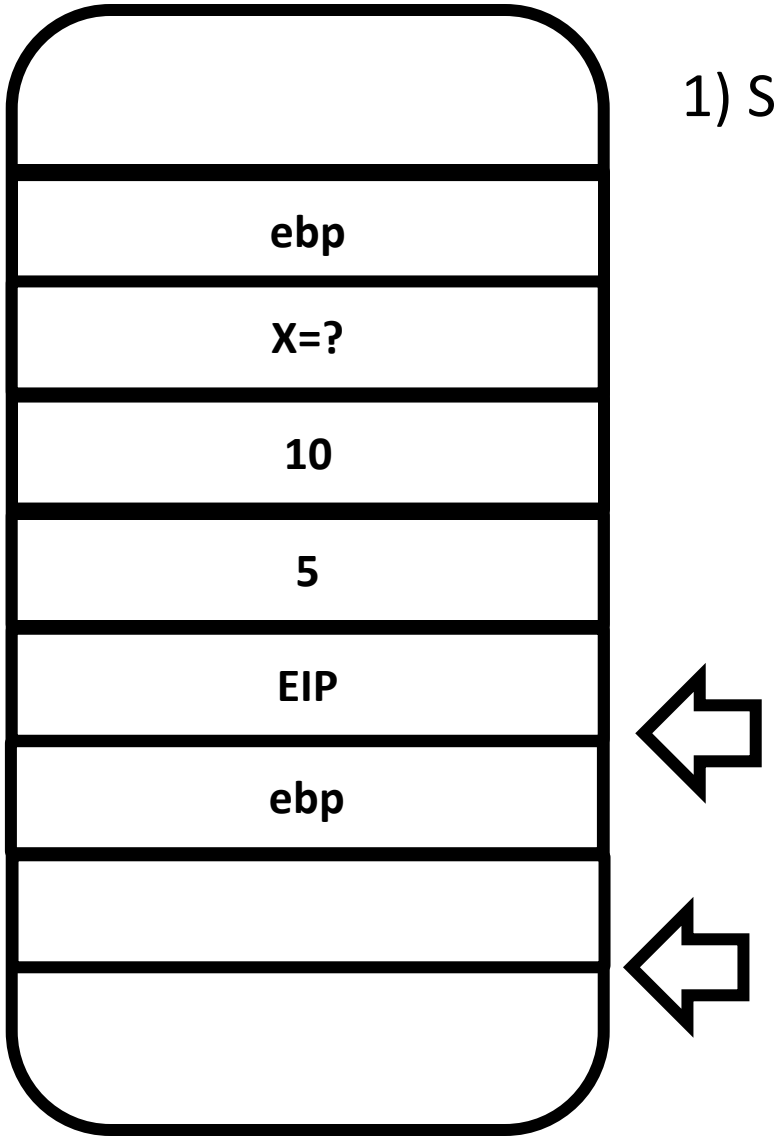


Call instruction pushes EIP to stack and jumps to add location

The Call Stack (Review)

```
int add(int a, int b) {
   int result = a+b;
   return result;
}
```

```
add:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
```



1) Save frame pointer and set to stack pointer

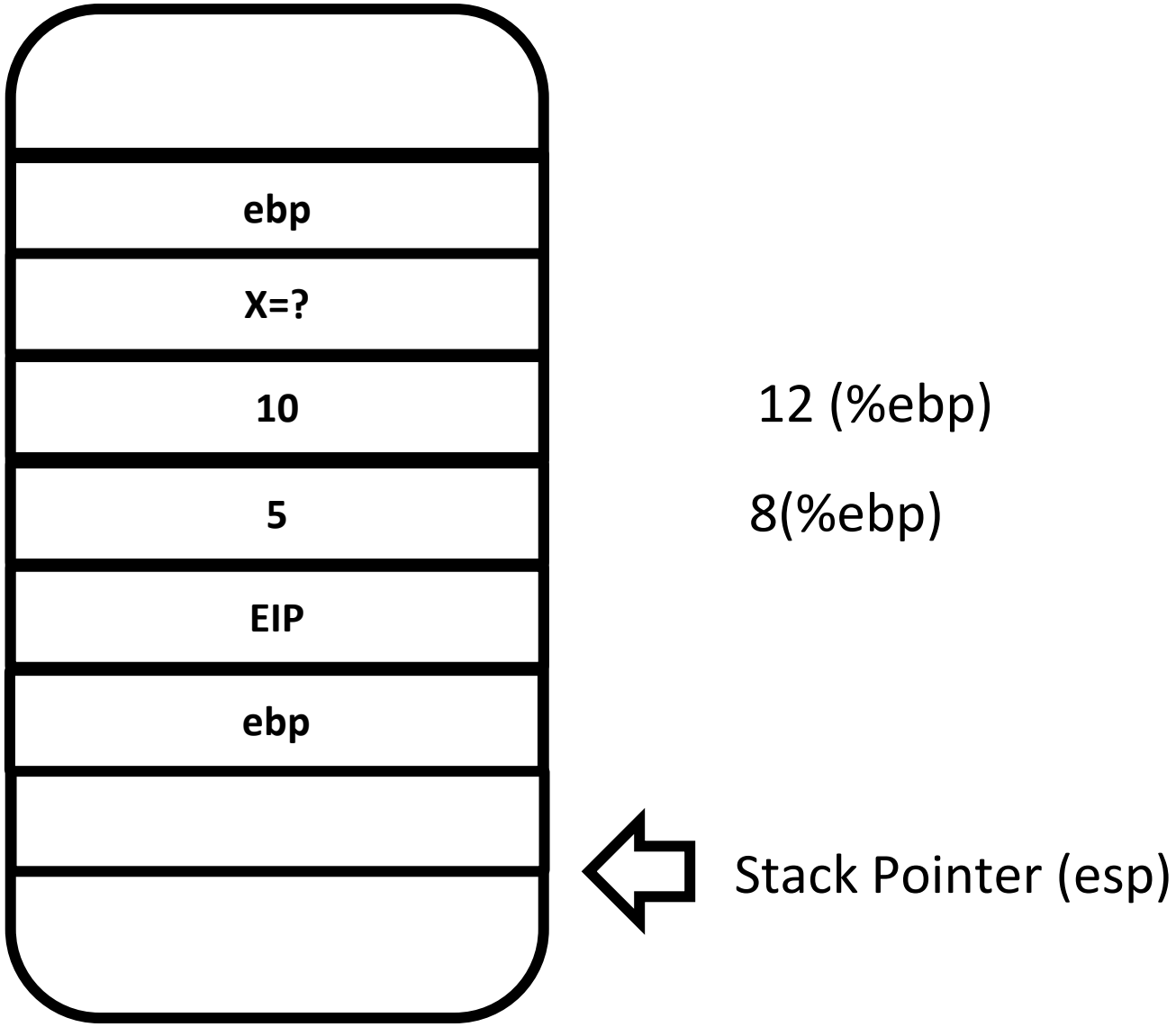
Stack Pointer (esp)

Stack Pointer (esp)

The Call Stack (Review)

```
int add(int a, int b) {
   int result = a+b;
   return result;
}
```

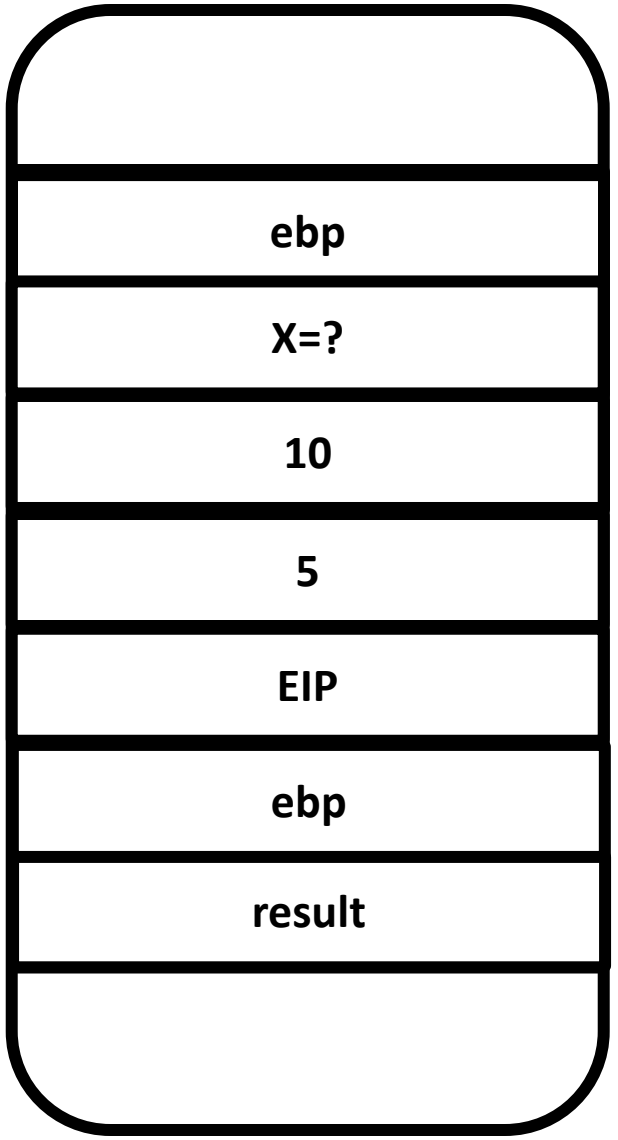
```
add:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
```



The Call Stack (Review)

```
int add(int a, int b) {
   int result = a+b;
   return result;
}
```

```
add:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    movl    %eax, -4(%ebp)
```

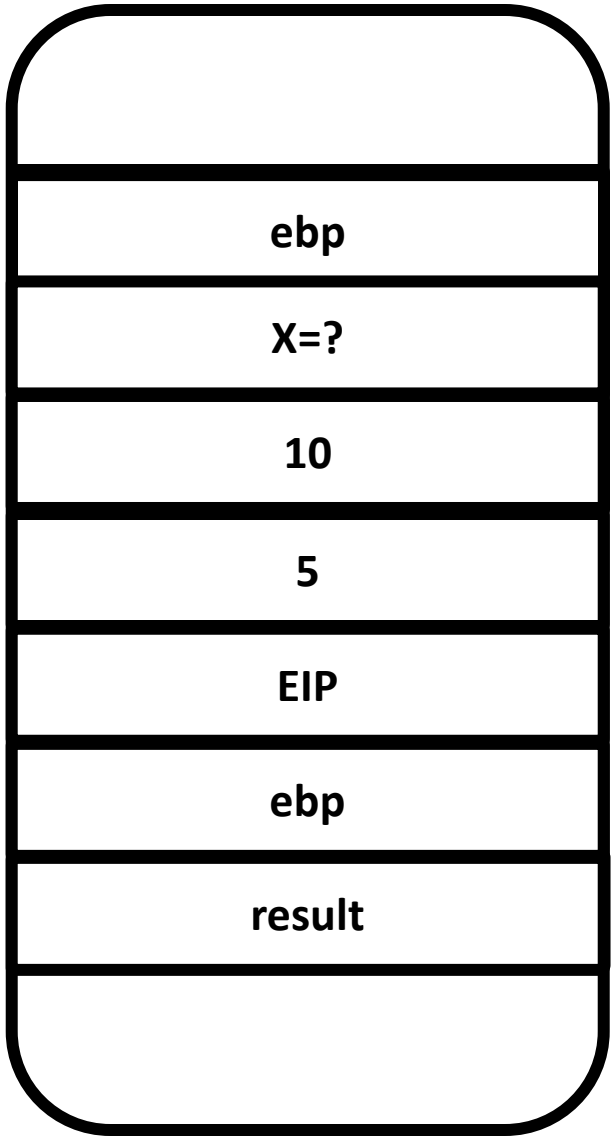


Local Variables are stored in the stack frame

The Call Stack (Review)

```
int add(int a, int b) {
   int result = a+b;
   return result;
}
```

```
add:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    movl    %eax, -4(%ebp)
    movl    -4(%ebp), %eax
```

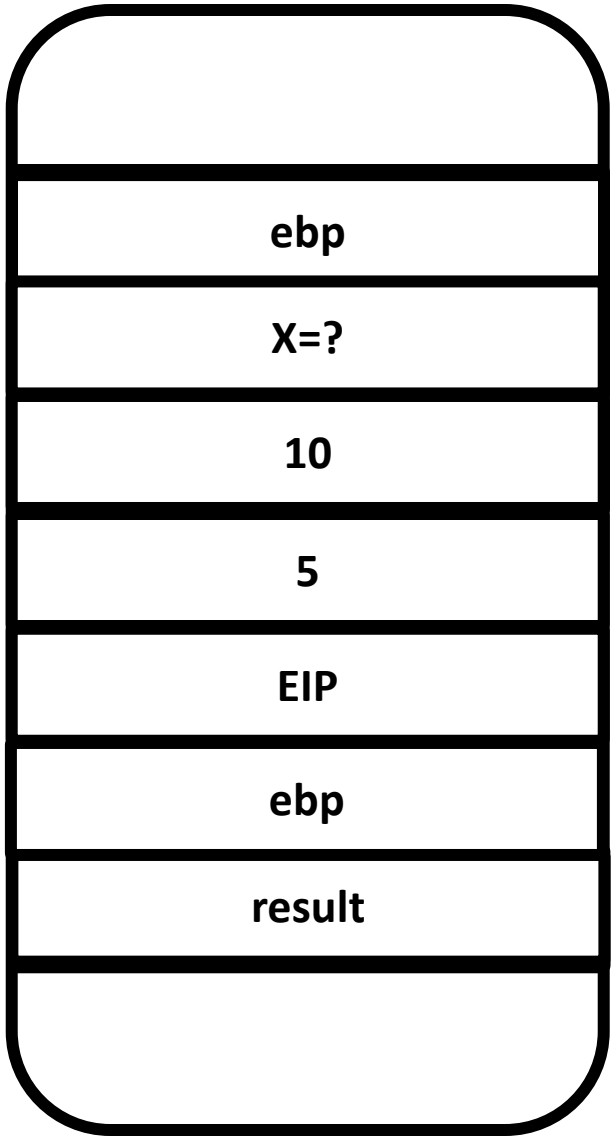


Move return value to eax register

The Call Stack (Review)

```
int add(int a, int b) {
   int result = a+b;
   return result;
}
```

```
add:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    movl    %eax, -4(%ebp)
    movl    -4(%ebp), %eax
    leave
    ret
```



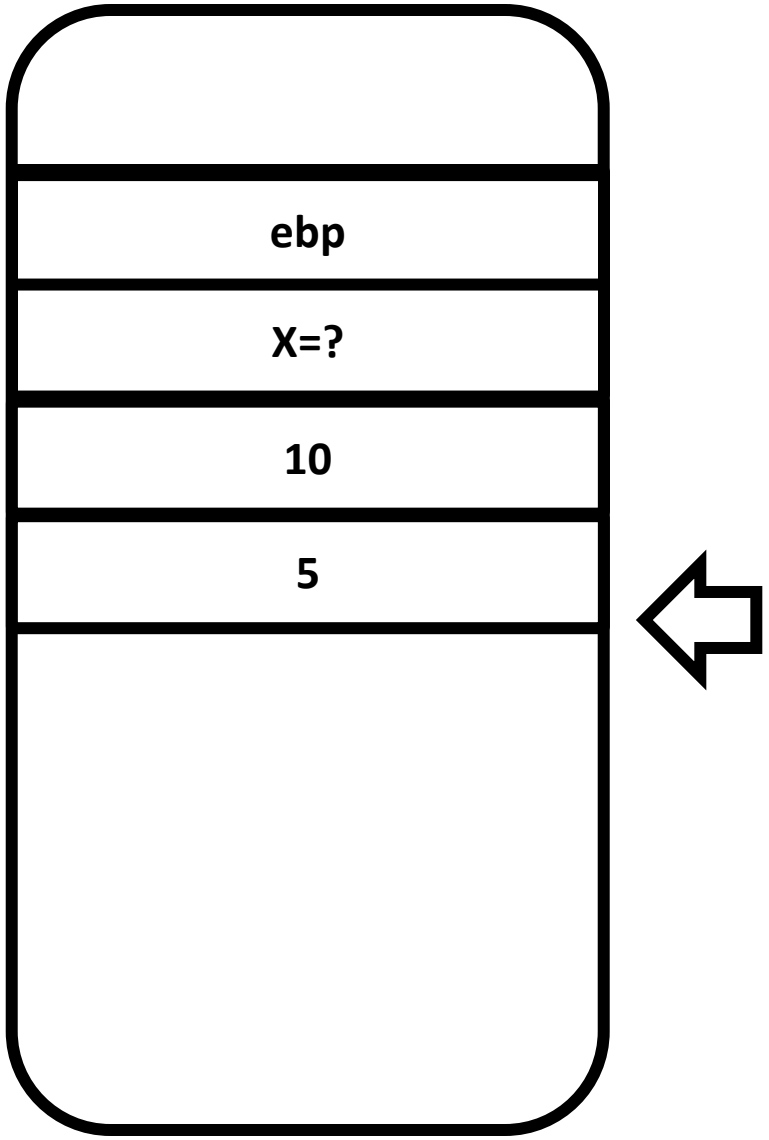
Leave instruction restores caller's frame (pops local variables and ebp)

Return instruction pops EIP and restores control to EIP

The Call Stack (Review)

```
void foo() {
  int x = add(5,10);
}
```

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        pushl   $10
        pushl   $5
        call    add
```



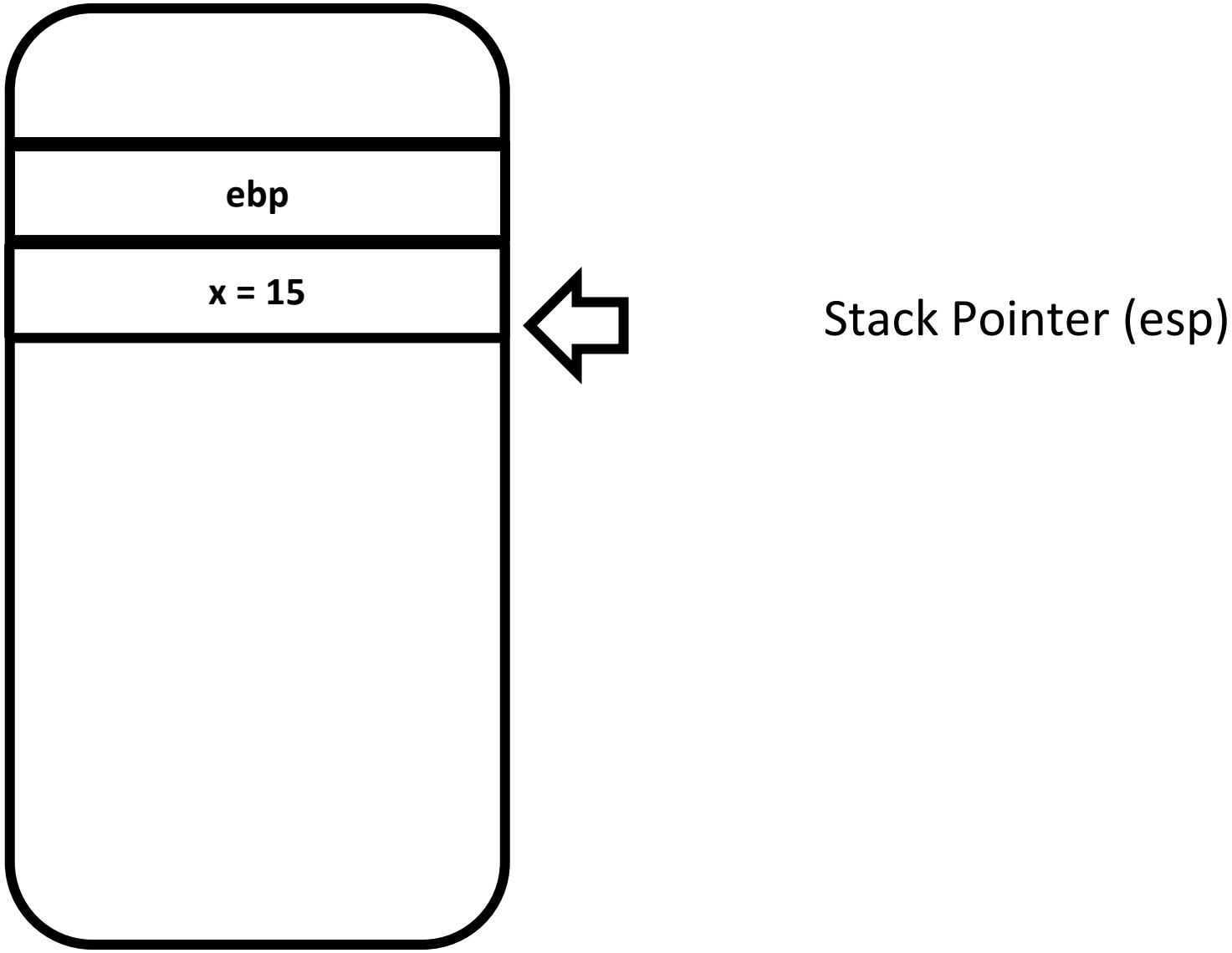
Pop function parameters

Stack Pointer (esp)

The Call Stack (Review)

```
void foo() {
  int x = add(5,10);
}
```

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        pushl   $10
        pushl   $5
        call    add
        movl    %eax, -4(%ebp)
```



Really Really Really Big Idea

The state of a program's execution is succinctly and completely represented by CPU register state

EIP, ESP, EBP, Eflags/PSW

Goal 2: User -> Kernel Mode

Goal 3: User -> Kernel Mode

Key Requirement:

Malicious user program (or IO device) cannot corrupt the kernel.

Interrupts, exceptions or system calls handled similarly

=> fewer code paths, fewer bugs.

1) Limited Entry

Cannot jump to arbitrary code in kernel

2) Atomic Switch

Switch from process stack to kernel stack

3) Transparent Execution

Restore prior state to continue program

Interrupt Handling Roadmap

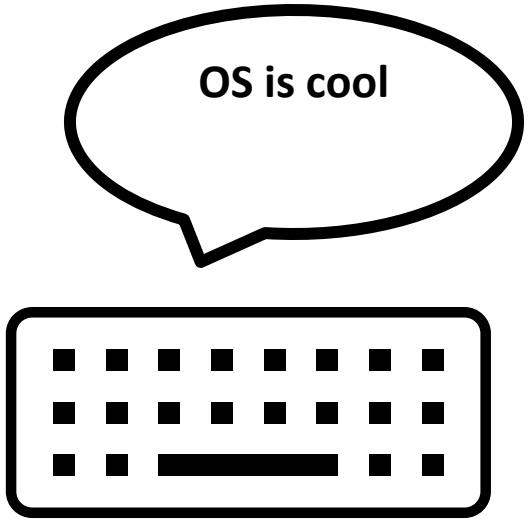
1) Processor detects interrupt

2) Suspend user program and switch to kernel stack

3) Identify interrupt type and invoke appropriate interrupt handler

4) Restore user program

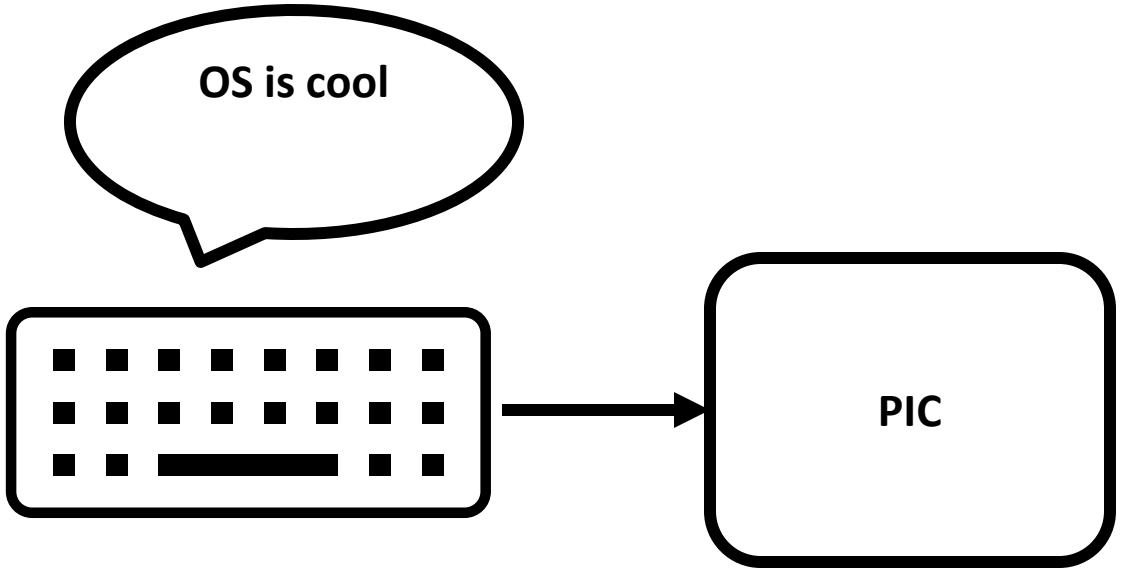
Don’t (Hardware) Interrupt Me




```
int add(int a, int b) {
  int result = a+b;
  return result;
}
```

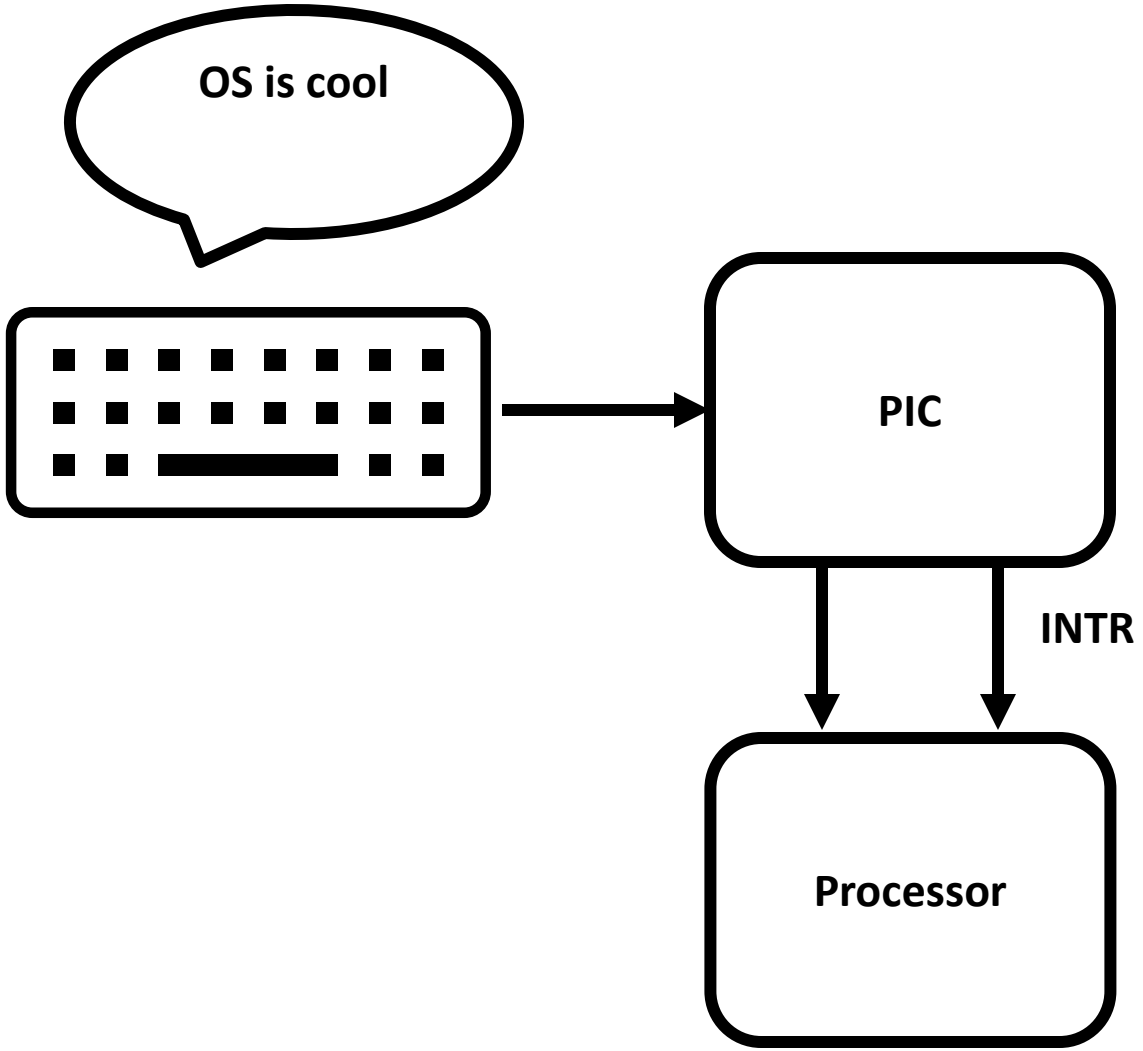
What happens when I type “OS is cool” on my keyboard while the Add program is running?

1) Interrupt Detection (Hardware)



Device sends electric signal interrupt request (IRQ) over interrupt request line to programmable interrupt controller (PIC)

1) Interrupt Detection (Hardware)



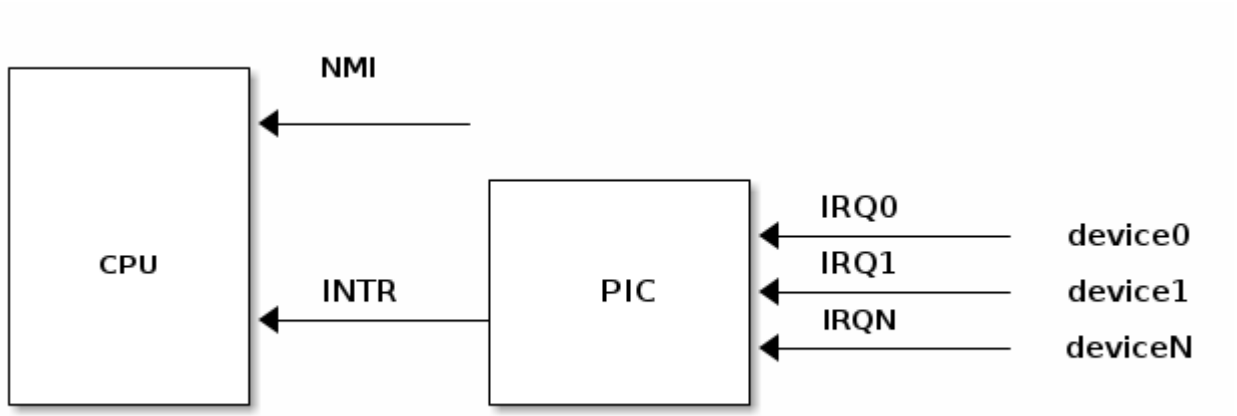
APIC converts IRQ to a vector number and sends signal to processor

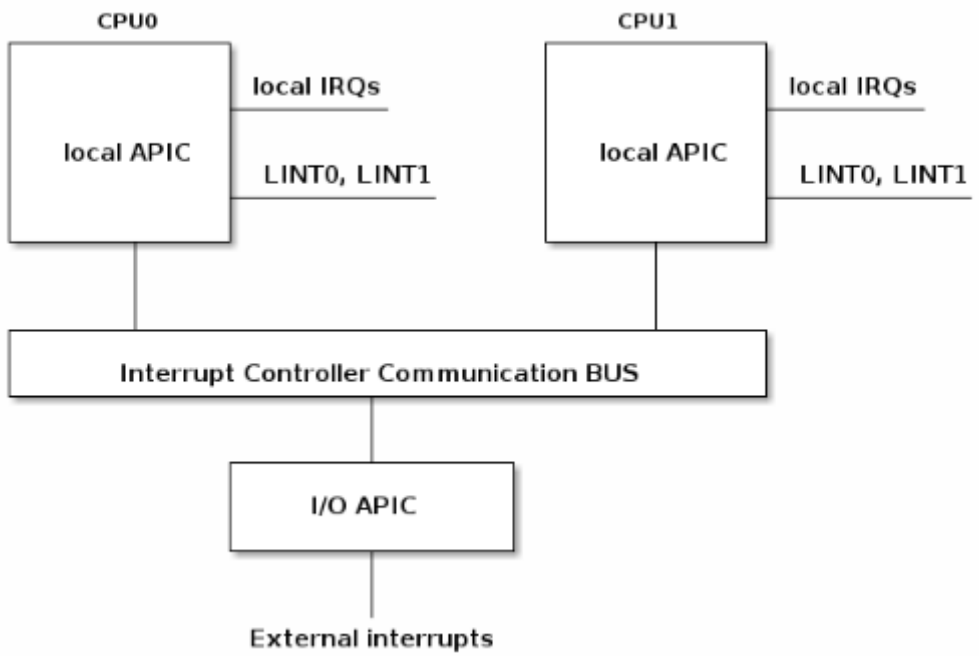
Processor detects interrupt

IRQs

IRQ	Bus type	Typically used by
00	none	Non-maskable Interrupt (NMI); system timer
01	none	Keyboard port
02	none	Programmable Interrupt Controller (PIC); cascade to IRQ 09
03	8/16-bit	Communications Port 2 (COM2:)
04	8/16-bit	Communications Port 1 (COM1:)
05	8/16-bit	Sound card; printer port (LPT2:)
06	8/16-bit	Floppy disk controller

Sidenote: PICS and APICs





2) Save Recovery State (Hardware)

Save register values (recovery state) for process recovery

```
int add(int a, int b) {
  int result = a+b;
  return result;
}
```

Which registers need to be saved by hardware to restore program?

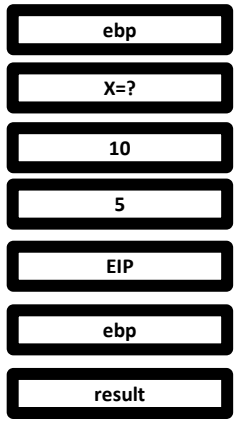
Stack Pointer (esp)

Program Counter (eip)

Execution Flags / Program Status Word (Eflags)

3) Switching (atomically) to Kernel Stack

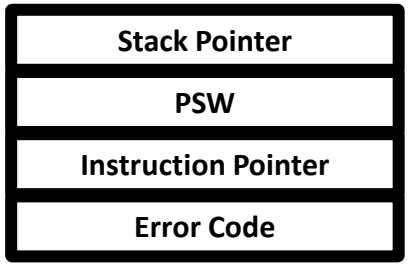
```
int add(int a, int b) {
  int result = a+b;
  return result;
}
```



User Stack

Switches stack pointer to base of kernel stack

Pushes recovery state onto the new stack
(+ optional error code)



Question 1:
Why did hardware need to save registers before switching to kernel stack?

Must overwrite EIP/SP when switching!

Question 2:
Why do we need a separate kernel stack?

Integrity and privacy concerns

A Tale of Two Stacks

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Xv6 Kernel (proc.h)

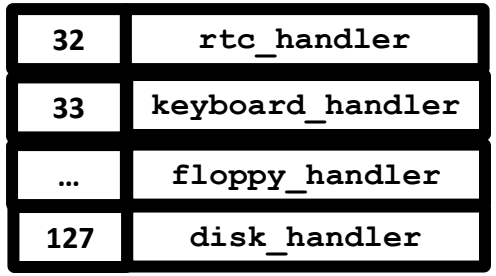
4) Invoke Interrupt Handler (Hardware)

Interrupt vector is an index into Interrupt Vector Table (or interrupt descriptor table).

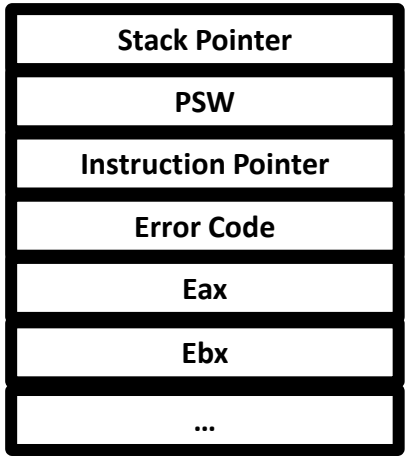
Index contains appropriate Interrupt Handler Routine

Control Unit sets EIP to handler

Handler saves all remaining user registers into stack and implements necessary logic (Transition software)



IDT Table in Linux

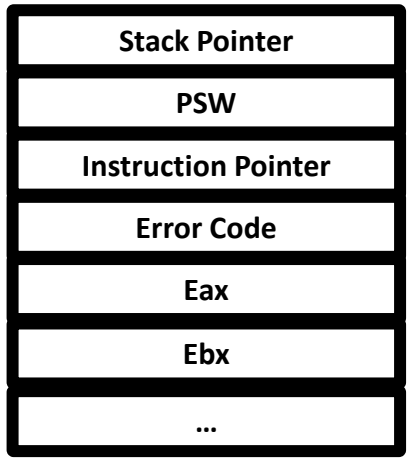


Kernel Stack

5) Return to Program

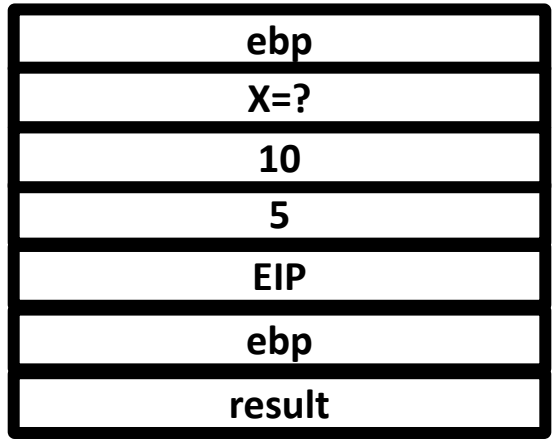
Pop all user registers from kernel stack (restore register state)

Invoke iret instruction to pop saved EIP, EFLAGS, and SP registers from kernel's exception stack to relevant registers



Kernel Stack

Return to user mode



User Stack

Concurrent Interrupts

What happens if an interrupt happens while processing an interrupt?

Hardware provides instruction to temporarily defer delivery of interrupt (disable interrupt), and re-enable them when safe (enable interrupt)

Interrupts are disabled when an interrupt handler is running

Periods during which interrupts are disabled should be very short!

Interrupt Summary

1) Device sends signal to APIC

2) Processor detects interrupt

3) Save Recovery State and switch to Kernel Stack

4) Jump to interrupt handler table at appropriate vector. Invoke interrupt handler

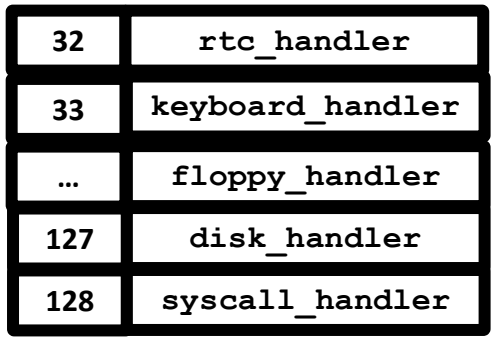
5) Restore user program

What about syscalls?

System calls are user functions that request services from the OS. Described as function call, with a name, parameters and return value.

Good news!
Syscalls are handled (almost) identically to interrupts.

What about syscalls?



Syscalls issue a "trap" instruction (`int 0x80`)
Generated interrupt will trigger exception vector 128!

How does handler know which syscall to execute?

System Call number fed in to %eax register.
System call number entry into *system call dispatch table,*

What about parameters and return values?
Propagated through registers.

Warning: Parameters must be carefully checked.

What about syscalls?

Four differences:

1) Extra-layer of indirection (system call table)

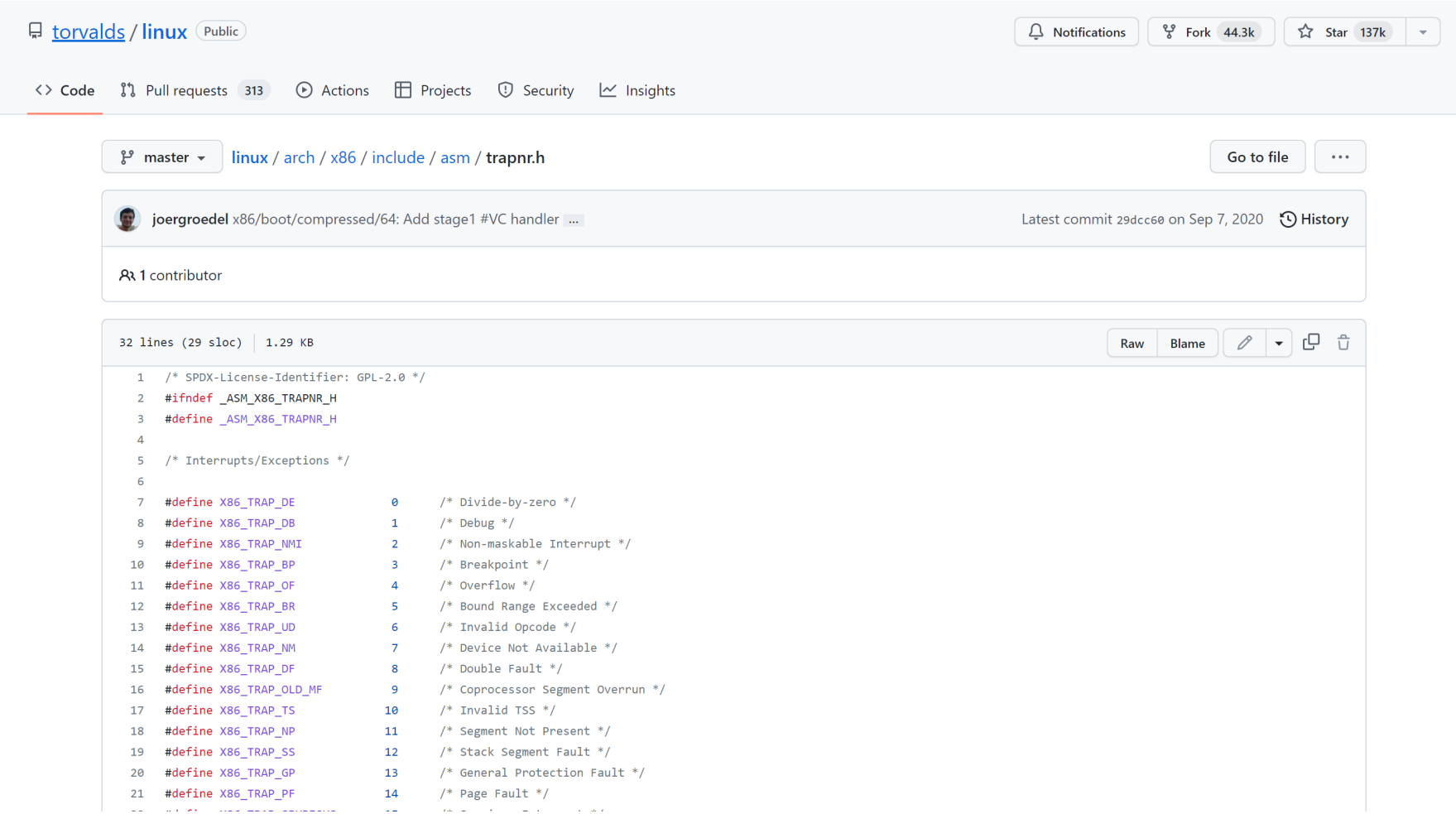
2) Leverage registers for parameters/values

3) When executing iret, increment EIP by one to go to next instruction

4) Usually, interrupts not disabled

What about exceptions?

It's the same!



torvalds / linux Public

Notifications | Fork 44.3k | Star 137k

Code | Pull requests 313 | Actions | Projects | Security | Insights

master | linux / arch / x86 / include / asm / trapnr.h | Go to file | ...

joergroedel x86/boot/compressed/64: Add stage1 #VC handler ... | Latest commit 29dcc60 on Sep 7, 2020 | History

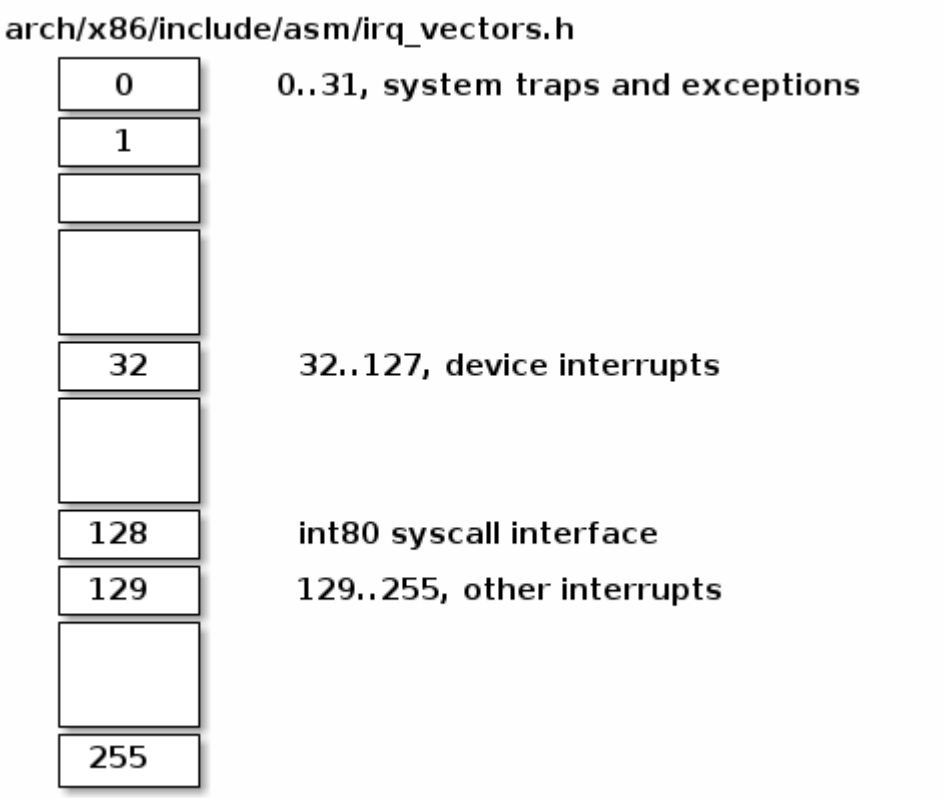
1 contributor

32 lines (29 sloc) | 1.29 KB | Raw | Blame

```
1  /* SPDX-License-Identifier: GPL-2.0 */
2  #ifndef _ASM_X86_TRAPNR_H
3  #define _ASM_X86_TRAPNR_H
4
5  /* Interrupts/Exceptions */
6
7  #define X86_TRAP_DE             0      /* Divide-by-zero */
8  #define X86_TRAP_DB             1      /* Debug */
9  #define X86_TRAP_NMI            2      /* Non-maskable Interrupt */
10 #define X86_TRAP_BP             3      /* Breakpoint */
11 #define X86_TRAP_OF             4      /* Overflow */
12 #define X86_TRAP_BR             5      /* Bound Range Exceeded */
13 #define X86_TRAP_UD             6      /* Invalid Opcode */
14 #define X86_TRAP_NM             7      /* Device Not Available */
15 #define X86_TRAP_DF             8      /* Double Fault */
16 #define X86_TRAP_OLD_MF         9      /* Coprocessor Segment Overrun */
17 #define X86_TRAP_TS            10      /* Invalid TSS */
18 #define X86_TRAP_NP            11      /* Segment Not Present */
19 #define X86_TRAP_SS            12      /* Stack Segment Fault */
20 #define X86_TRAP_GP            13      /* General Protection Fault */
21 #define X86_TRAP_PF            14      /* Page Fault */
```

The magic of the IVT

Single, well-defined entry point in the kernel helps with security



Browse the source code of linux/arch/x86/include/asm/irq_vectors.h

```
1  /* SPDX-License-Identifier: GPL-2.0 */
2  #ifndef _ASM_X86_IRQ_VECTORS_H
3  #define _ASM_X86_IRQ_VECTORS_H
4
5  #include <linux/threads.h>
6  /*
7   * Linux IRQ vector layout.
8   *
9   * There are 256 IDT entries (per CPU - each entry is 8 bytes) which can
10  * be defined by Linux. They are used as a jump table by the CPU when a
11  * given vector is triggered - by a CPU-external, CPU-internal or
12  * software-triggered event.
13  *
14  * Linux sets the kernel code address each entry jumps to early during
15  * bootup, and never changes them. This is the general layout of the
16  * IDT entries:
17  *
18  *  Vectors   0 ...  31 : system traps and exceptions - hardcoded events
19  *  Vectors  32 ... 127 : device interrupts
20  *  Vector  128         : legacy int80 syscall interface
21  *  Vectors 129 ... INVALIDATE_TLB_VECTOR_START-1 except 204 : device interrupts
22  *  Vectors INVALIDATE_TLB_VECTOR_START ... 255 : special interrupts
23  *
24  * 64-bit x86 has per CPU IDT tables, 32-bit has one shared IDT table.
25  *
26  * This file enumerates the exact layout of them:
27  */
28
```

Tension between performance and simplicity

Accessing IDT can be slow if not in cache.
Syscalls very common, can we make them cheaper?

Allocate a special register (machine specific register) to directly store address of system call dispatch table

Store register call in the rax register

But backwards compatibility …

Goals for today

- (Continued) Hardware support for dual mode

  Privileged Instructions, Memory Isolation, Timer Interrupts, Safe Context Switching.

- 61C Review: The Stack

  Stack Pointer, Frame Pointer, Program Counter

- How to switch from user mode to kernel mode and back?

  Switch to specified location in kernel & atomic.

  Interrupts, Syscalls, Exceptions handled identically. Use of the interrupt vector table