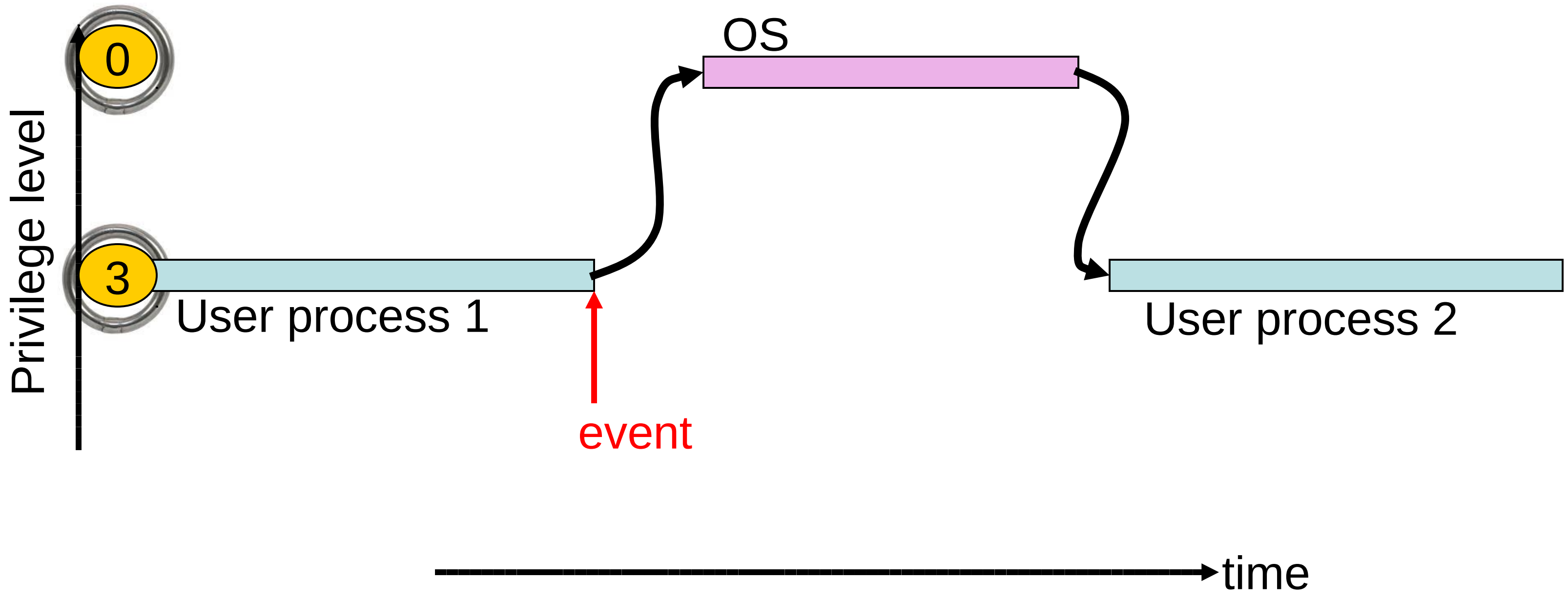
Interrupts, Exceptions, and System Calls

Chester Rebeiro

IIT Madras

OS & Events

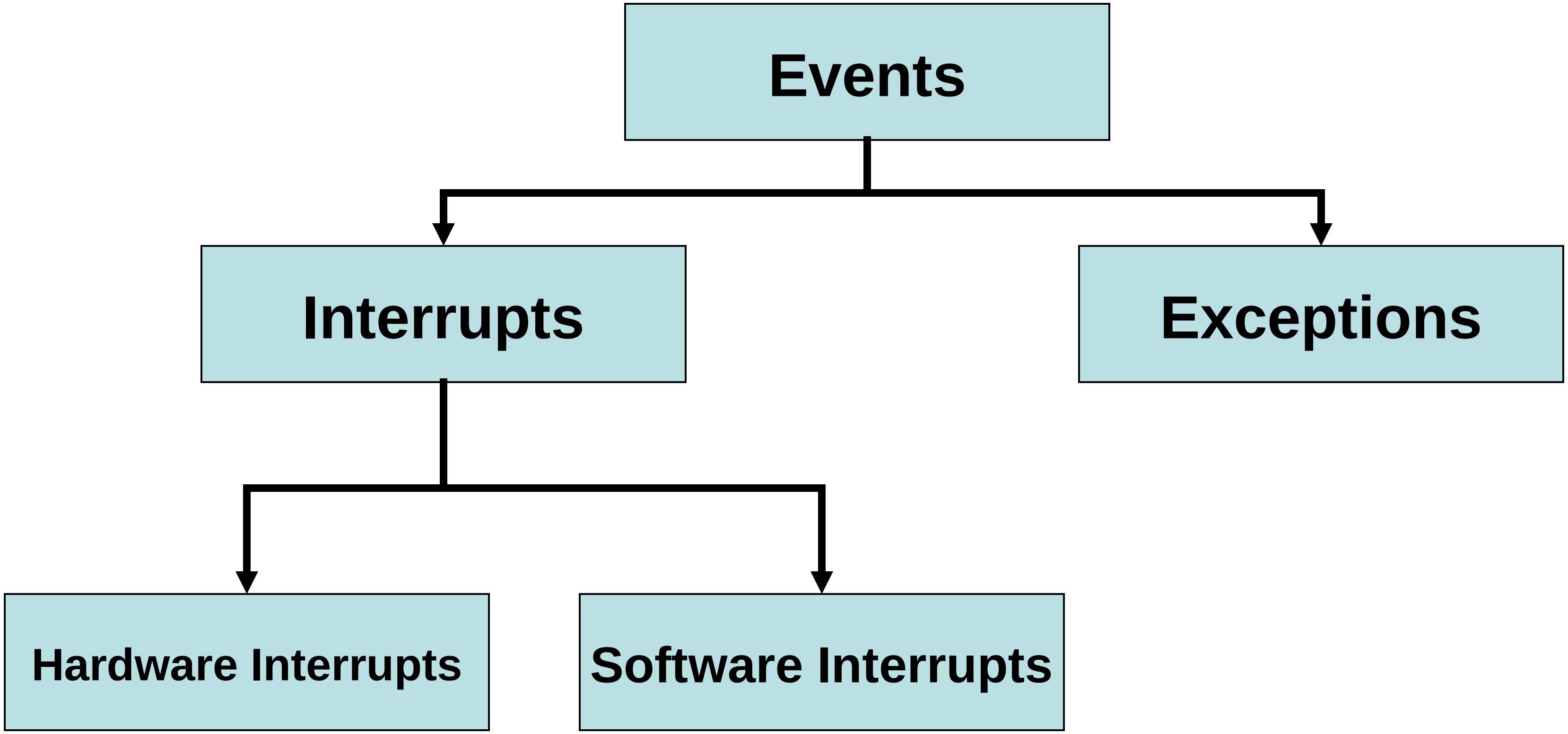
- OS is event driven
  - – i.e. executes only when there is an interrupt, trap, or system call



Why event driven design?

- OS cannot trust user processes
  - User processes may be buggy or malicious
  - User process crash should not affect OS
- OS needs to guarantee fairness to all user processes
  - One process cannot ‘hog’ CPU time
  - Timer interrupts

Event Types

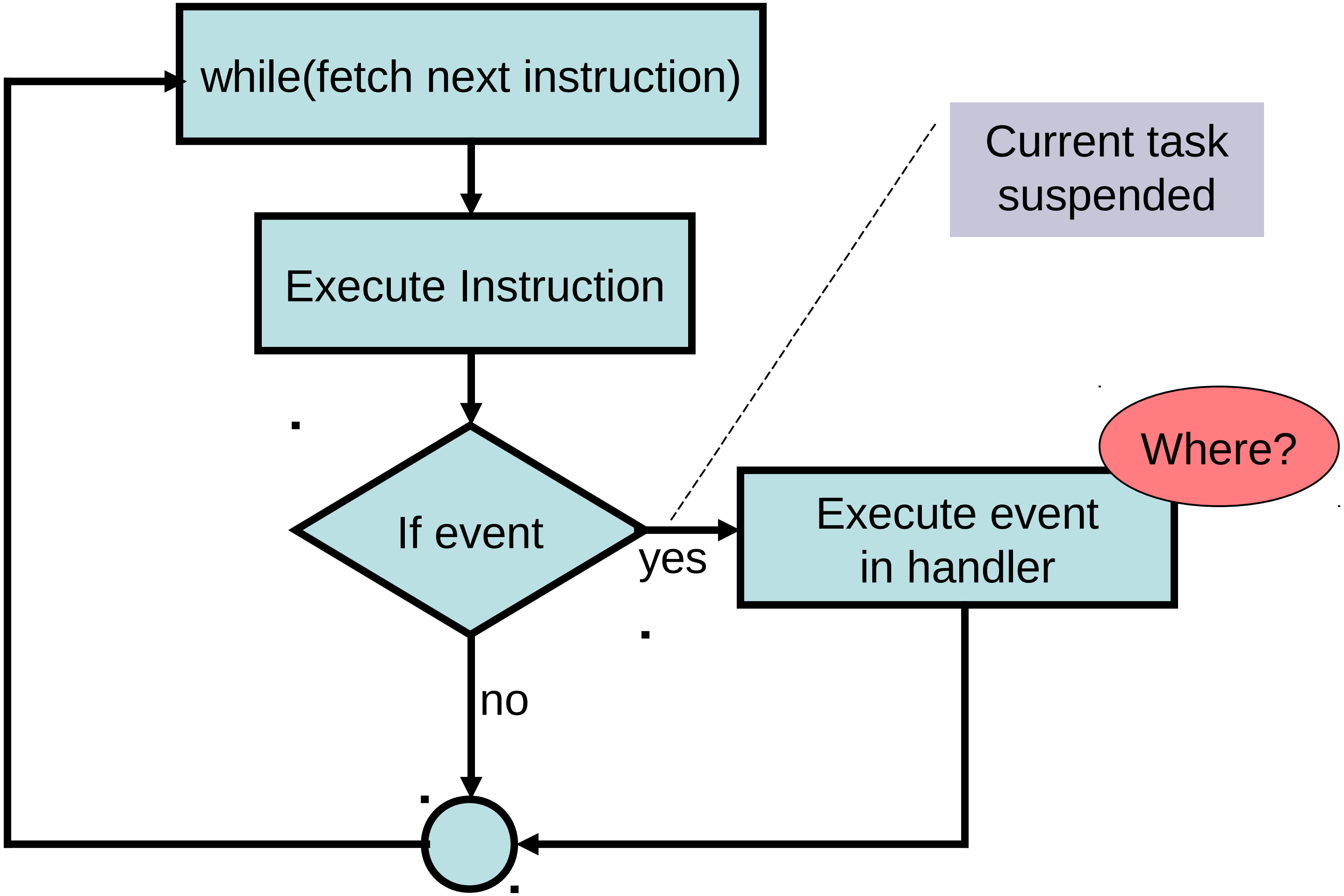

Events
Interrupts
Exceptions
Hardware Interrupts
Software Interrupts

Events

- Interrupts : raised by hardware or programs to get OS attention
  - Types
    - Hardware interrupts : raised by external hardware devices
    - Software Interrupts : raised by user programs

- Exceptions : due to illegal operations

Event view of CPU



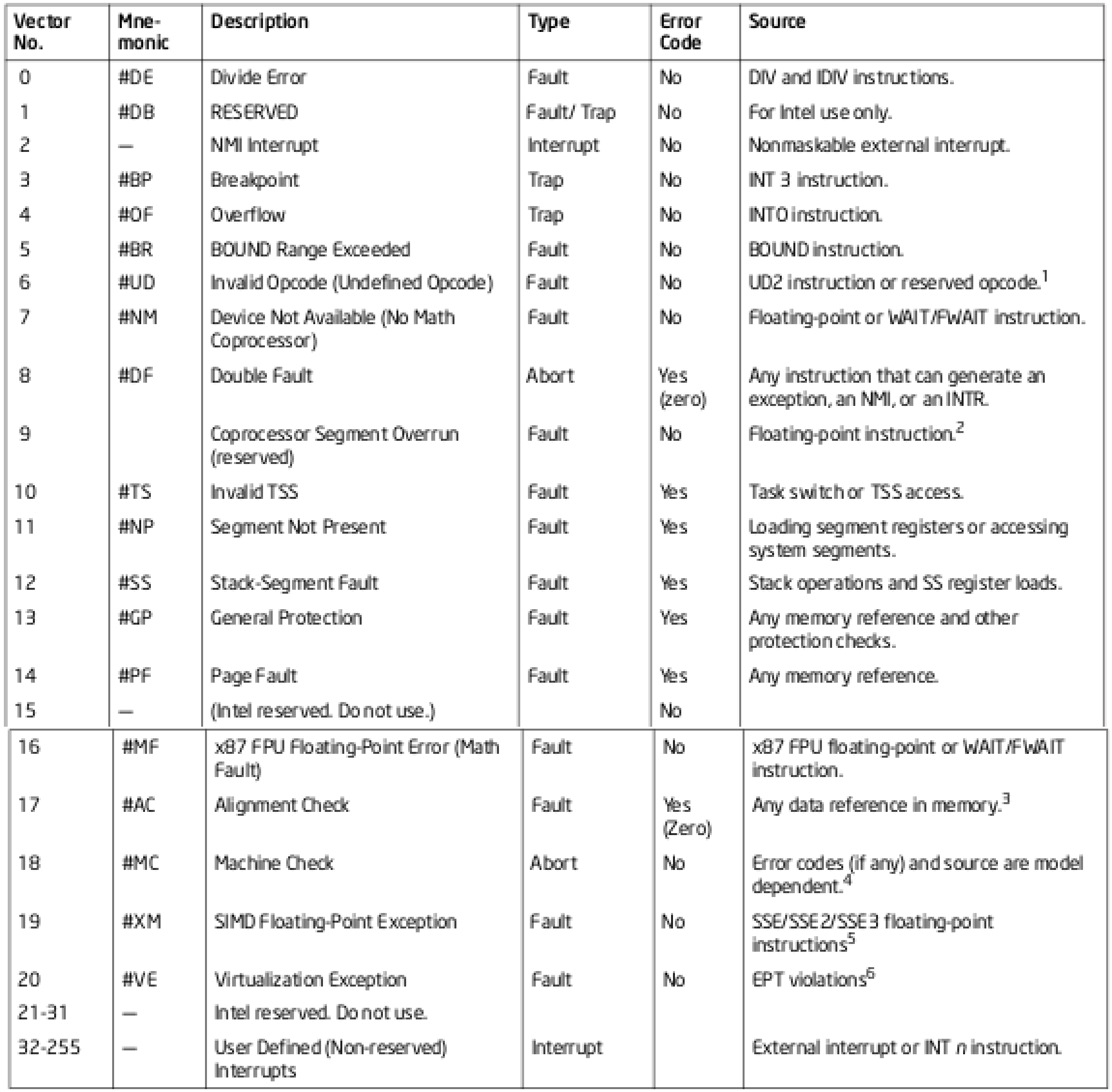
Exception & Interrupt Vectors

Event occured

What to execute next?

- Each interrupt/exception provided a number
- Number used to index into an Interrupt descriptor table (IDT)
- IDT provides the entry point into a interrupt/exception handler
- 0 to 255 vectors possible
  - 0 to 31 used internally
  - Remaining can be defined by the OS

Exception and Interrupt Vectors



Vector No.	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	RESERVED	Fault/ Trap	No	For Intel use only.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[5]
20	#VE	Virtualization Exception	Fault	No	EPT violations[6]
21-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

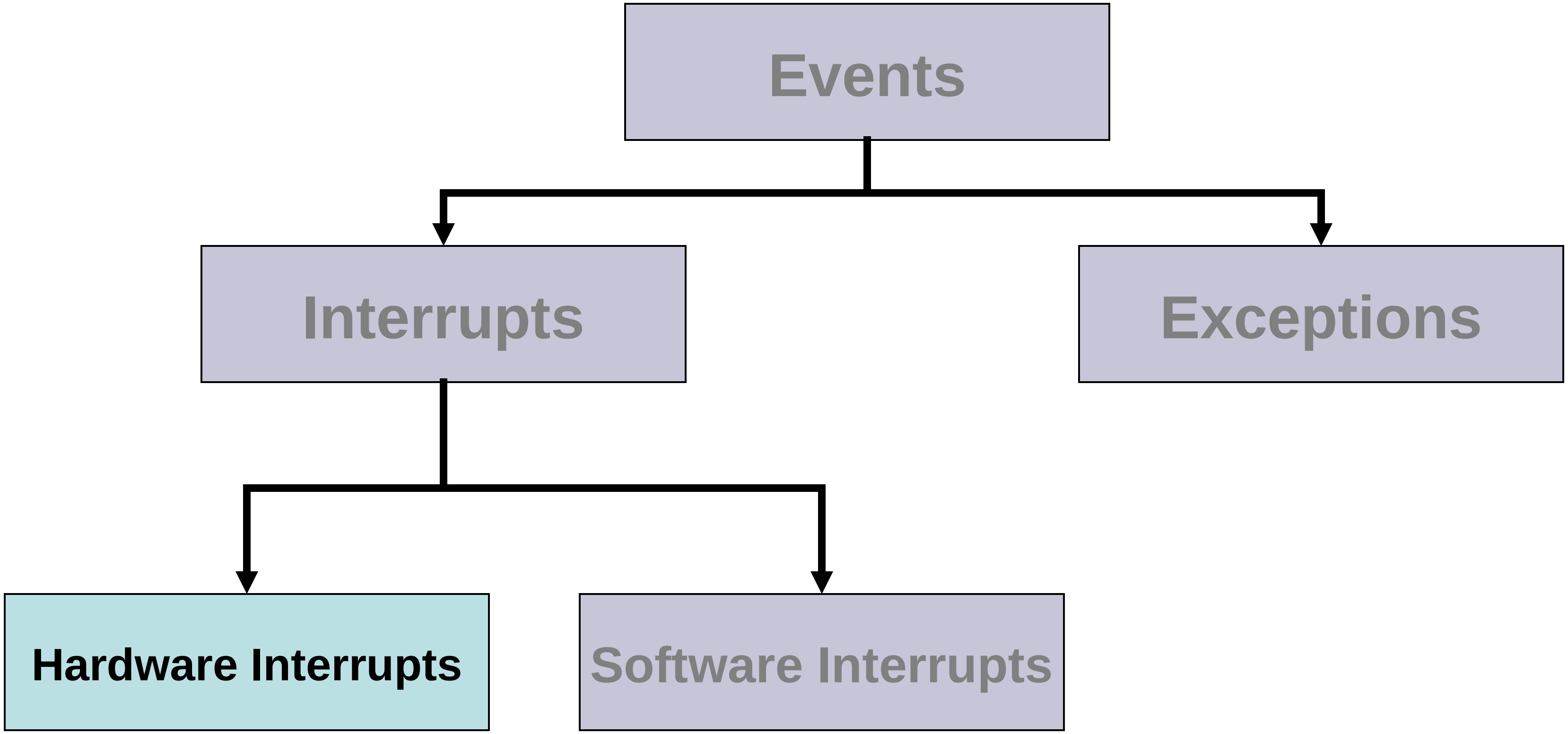
xv6 Interrupt Vectors

- 0 to 31 reserved by Intel
- 32 to 63 used for hardware interrupts

  T_IRQ0 = 32 (added to all hardware IRQs to scale them)
- 64 used for system call interrupt

ref : traps.h ([31], 3152)

Events


Events
Interrupts
Exceptions
Hardware Interrupts
Software Interrupts

Why Hardware Interrupts?

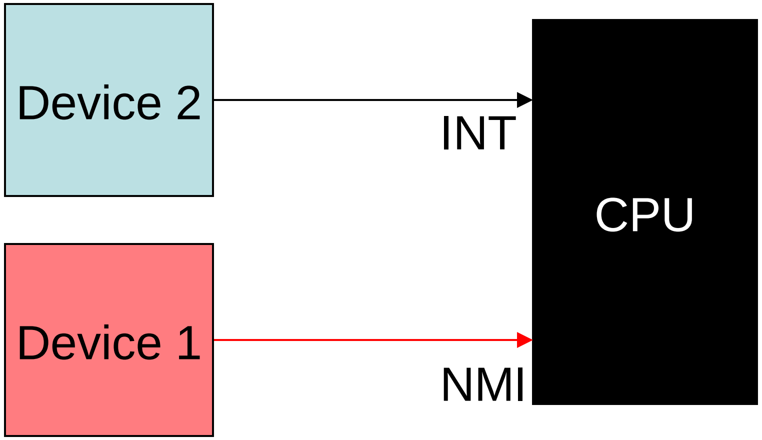
- Several devices connected to the CPU
  - eg. Keyboards, mouse, network card, etc.
- These devices occasionally need to be serviced by the CPU
  - eg. Inform CPU that a key has been pressed
- These events are asynchronous i.e. we cannot predict when they will happen.
- Need a way for the CPU to determine when a device needs attention

Possible Solution : Polling

- CPU periodically queries device to determine if they need attention
- Useful when device often needs to send information
  - For example in data acquisition systems
- If device does not need attention often,
  - Polling wastes CPU time

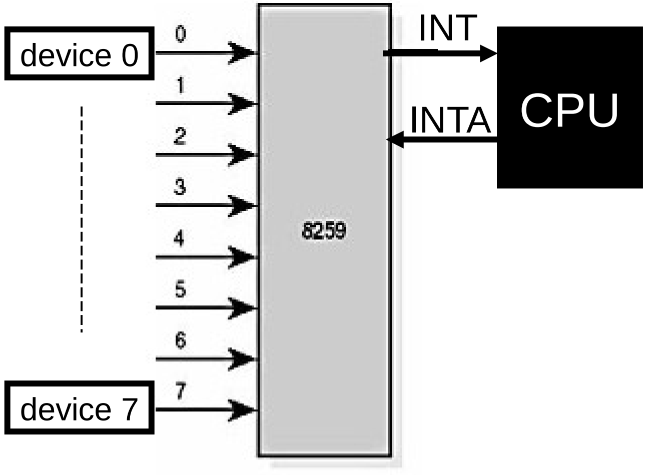
Interrupts

- Each device signals to the CPU that it wants to be serviced
- Generally CPUs have 2 pins
  - INT : Interrupt
  - NMI : Non maskable – for very critical signals
- How to support more than two interrupts?



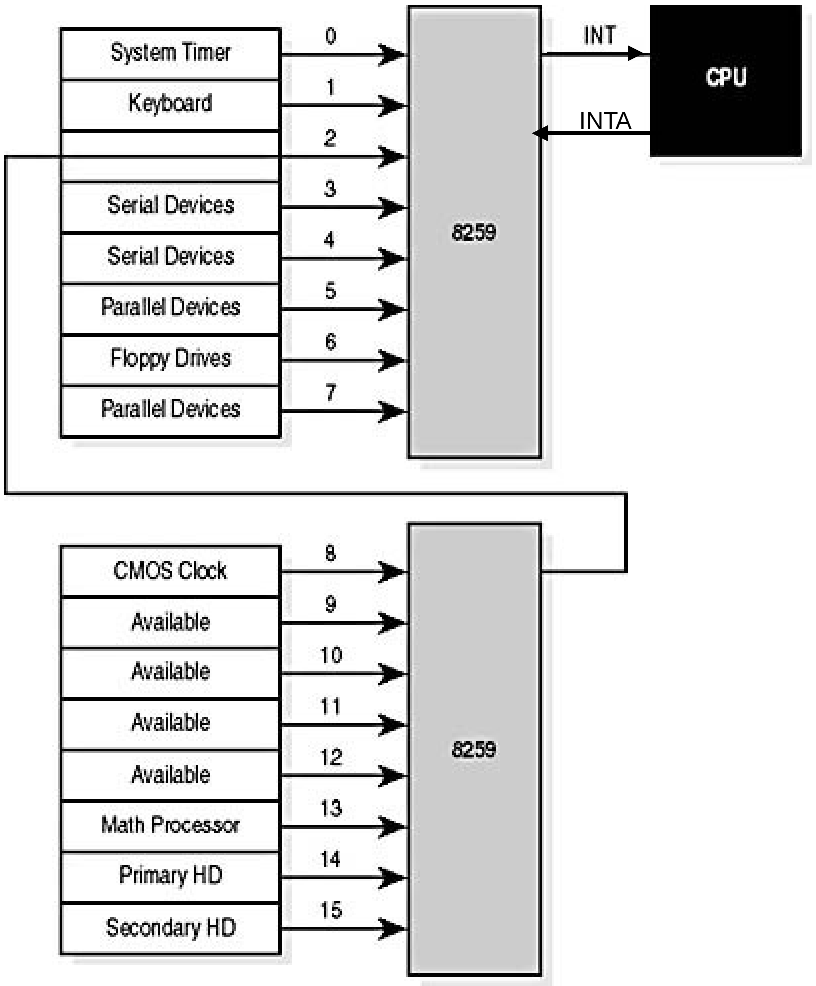
8259 Programmable Interrupt Controller

- 8259 (Programmable interrupt controller) relays upto 8 interrupt to CPU
- Devices raise interrupts by an ‘interrupt request’ (IRQ)
- CPU acknowledges and queries the 8259 to determine which device interrupted
- Priorities can be assigned to each IRQ line
- 8259s can be cascaded to support more interrupts



Interrupts in legacy CPUs

- 15 IRQs (IRQ0 to IRQ15), so 15 possible devices
- Interrupt types
  - Edge
  - Level
- Limitations
  - Limited IRQs
  - Spurious interrupts by 8259
    - Eg. de-asserted IRQ before IRQA



Edge vs Level Interrupts

- Level triggered Interrupt : as long as the IRQ line is asserted you get an interrupt.
  - Level interrupt still active even after interrupt service is complete
  - Stopping interrupt would require physically deactivating the interrupt
- Edge triggered Interrupt : Exactly one interrupt occurs when IRQ line is asserted
  - To get a new interrupt, the IRQ line must become inactive and then become active again
- Active high interrupts: When asserted, IRQ line is high (logic 1)

Edge vs Level Interrupts (the crying baby… an analogy)

- Level triggered interrupt :
  - when baby cries (interrupt) stop what you are doing and feed the baby
  - then put the baby down
  - if baby still cries (interrupt again) continue feeding
- Edge triggered interrupt
  - eg*. Baby cry monitor*, where light turns red when baby is crying. The light is turned off by a push button switch
    - if baby cries and stops immediately you see that the baby has cried (level triggered would have missed this)
    - if the baby cries and you press the push buttton, the light turns off, and remains off even though the button is pressed

http://venkateshabbarapu.blogspot.in/2013/03/edge-triggered-vs-level-triggered.html

Spurious Interrupts

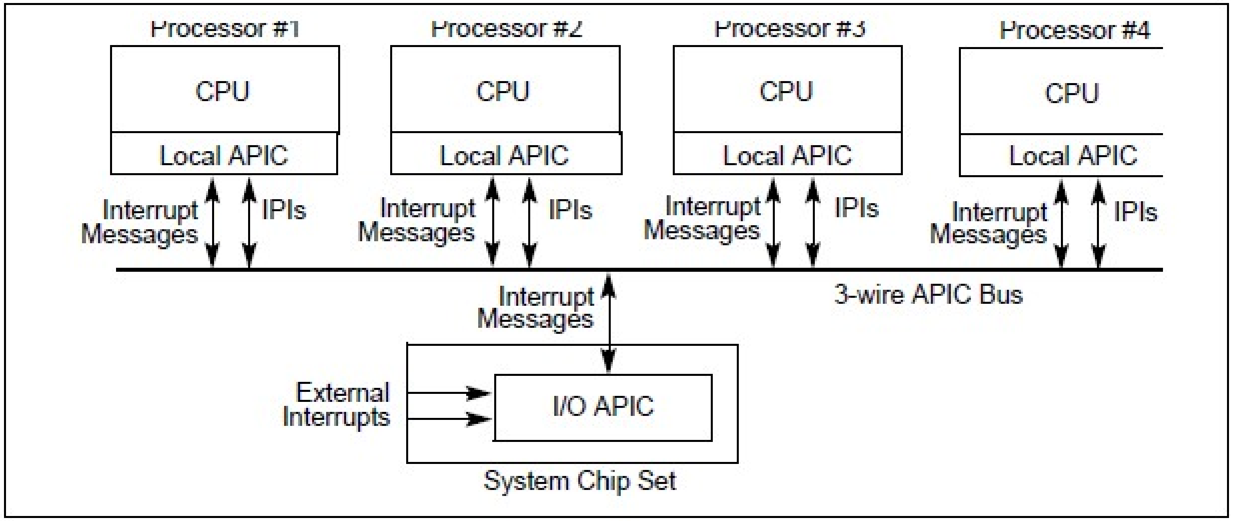
Consider the following Sequence

1. Device asserts level triggered interrupt
2. PIC tells CPU that there is an interrupt
3. CPU acknowledges and waits for PIC to send interrupt vector
4. However, device de-asserts interrupt. What does the PIC do?

This is a spurious interrupt

To prevent this, PIC sends a fake vector number called the spurious IRQ. This is the lowest priority IRQ.

Advanced Programmable Interrupt Controller (APIC)



- External interrupts are routed from peripherals to CPUs in multi processor systems through APIC
- APIC distributes and prioritizes interrupts to processors
- Interrupts can be configured as edge or level triggered
- Comprises of two components
  - Local APIC (LAPIC)
  - I/O APIC
- APICs communicate through a special 3-wire APIC bus.
  - In more recent processors, they communicate over the system bus

LAPIC and I/OAPIC

- LAPIC :
  - Receives interrupts from I/O APIC and routes it to the local CPU
  - Can also receive local interrupts (such as from thermal sensor, internal timer, etc)
  - Send and receive IPIs (Inter processor interrupts)
    - IPIs used to distribute interrupts between processors or execute system wide functions like booting, load distribution, etc.
- I/O APIC
  - Present in chipset (north bridge)
  - Used to route external interrupts to local APIC

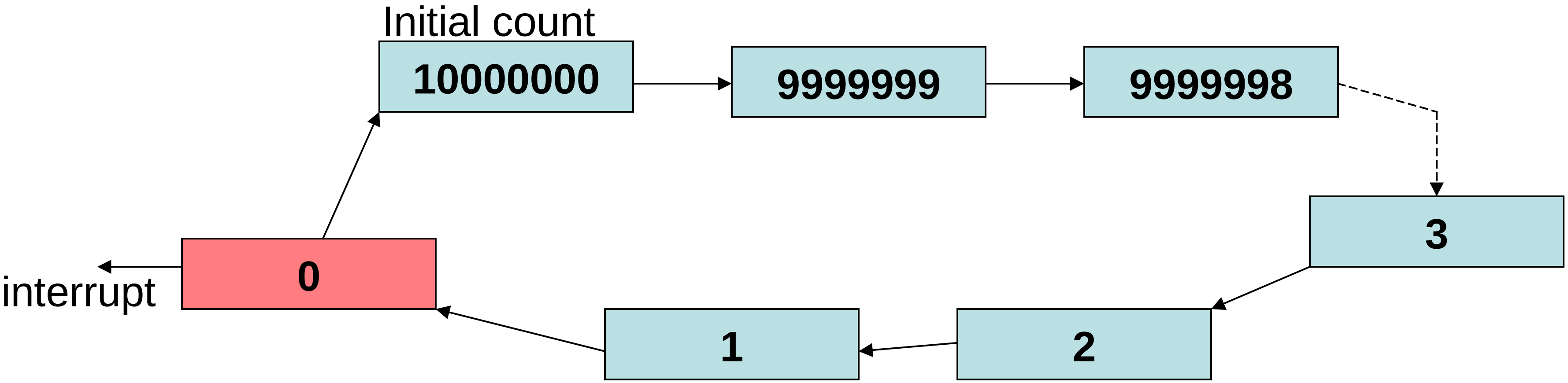
I/O APIC Configuration in xv6

- IO APIC : 82093AA I/O APIC
- Function : ioapicinit (in ioapic.c)
- All interrupts configured during boot up as
  - Active high
  - Edge triggered
  - Disabled (interrupt masked)
- Device drivers selectively turn on interrupts using ioapicenable
  - Three devices turn on interrupts in xv6
    - UART (uart.c)
    - IDE (ide.c)
    - Keyboard (console.c)

ref : ioapic.c [73], (http://www.intel.com/design/chipsets/datashts/29056601.pdf)

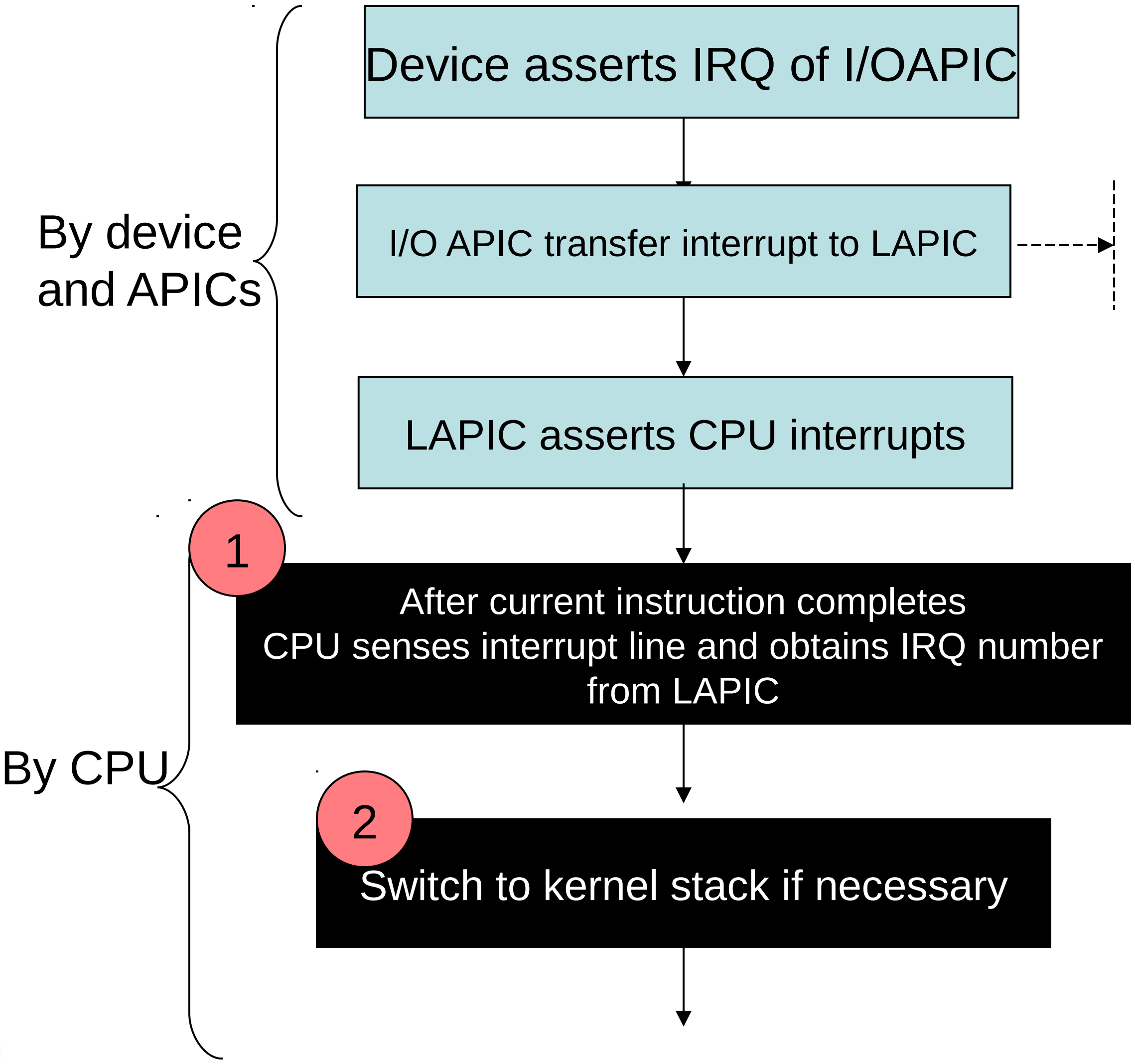
LAPIC Configuration in xv6

1. Enable LAPIC and set the spurious IRQ (i.e. the default IRQ)
2. Configure Timer
   - Initialize timer register (10000000)
   - Set to periodic



ref : lapic.c (lapicinit) (7151)

What happens when there is an Interrupt?



By device and APICs

Device asserts IRQ of I/OAPIC

I/O APIC transfer interrupt to LAPIC

LAPIC asserts CPU interrupts

By CPU

1

After current instruction completes
CPU senses interrupt line and obtains IRQ number from LAPIC

2

Switch to kernel stack if necessary

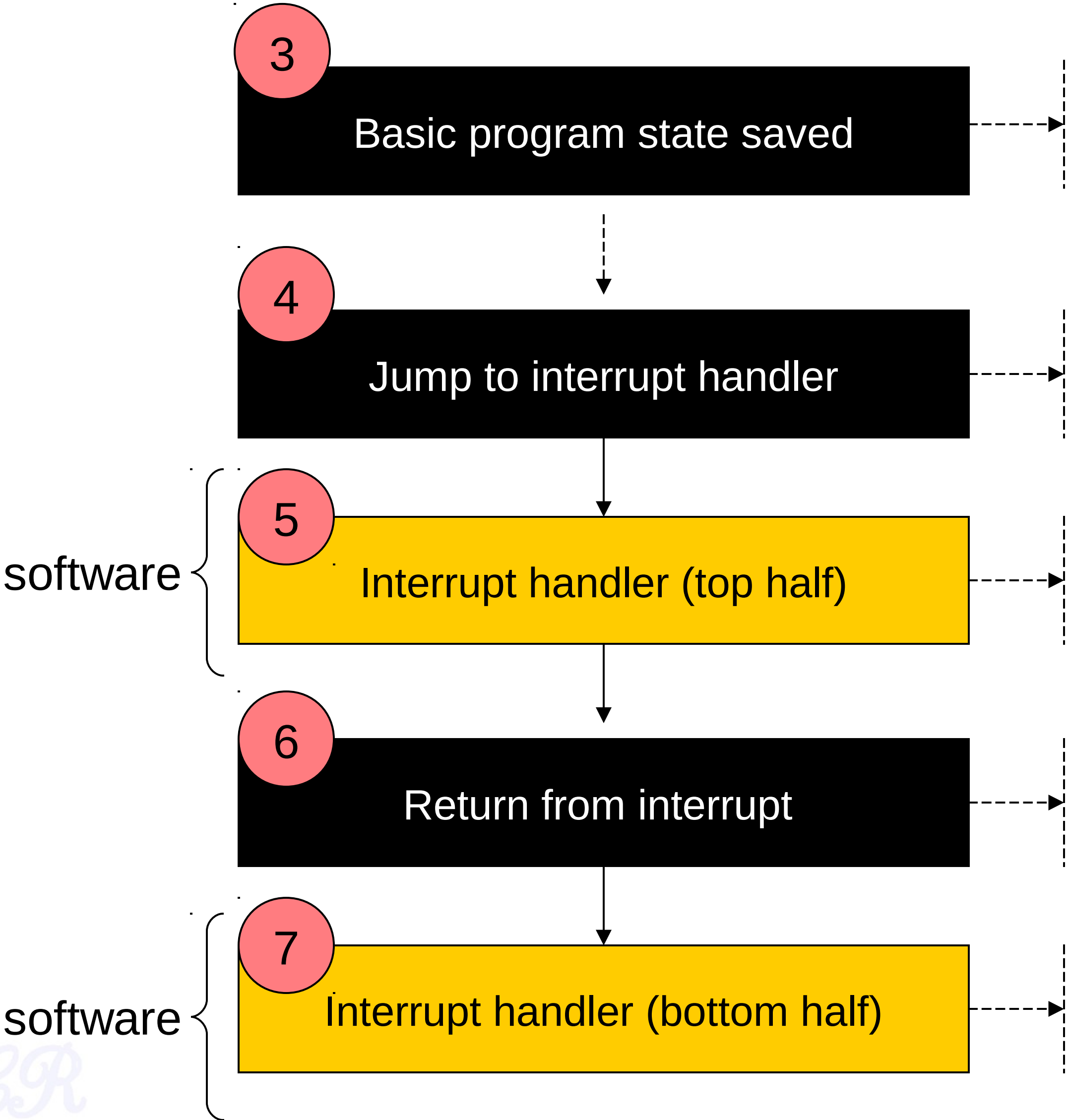
Either special 3 wire APIC bus
system bus

🟦 By device and APICs

⬛ Done by CPU automaticall
y

🟨 Done in software

What more happens when there is an Interrupt?



3 Basic program state saved

X86 saves the SS, ESP, EFLAGS, CS, EIP, error code on stack (restored by iret instruction). Suspends current task.

4 Jump to interrupt handler

How does hardware find the OS interrupt handler?

software

5 Interrupt handler (top half)

Just do the important stuff like
… respond to interrupt
… more storing of program state
… schedule the bottom half
… IRET

6 Return from interrupt

Restore flags and registers saved earlier. Restore running task.

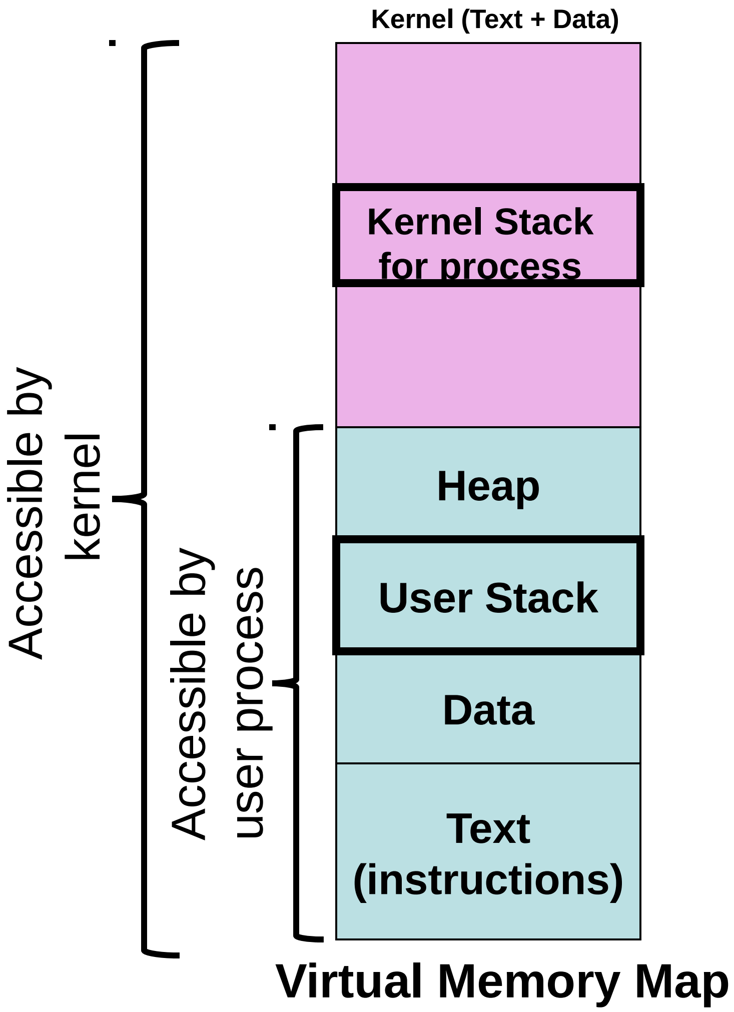
software

7 Interrupt handler (bottom half)

The work horse for the interrupt

Stacks

- Each process has two stacks
  - – a user space stack
  - – a kernel space stack



2

Switching Stack (to switch or not to switch)

- When event occurs OS executes
  - If executing user process, privilege changes from low to high
  - If already in OS no privilege change
- Why switch stack?
  - OS cannot trust stack (SS and ESP) of user process
  - Therefore stack switch needed only when moving from user to kernel mode
- How to switch stack?
  - CPU should know locations of the new SS and ESP.
  - Done by task segment descriptor

Done automatically by CPU

To Switch or not to Switch

Executing in Kernel space

- No stack switch
- Use the current stack

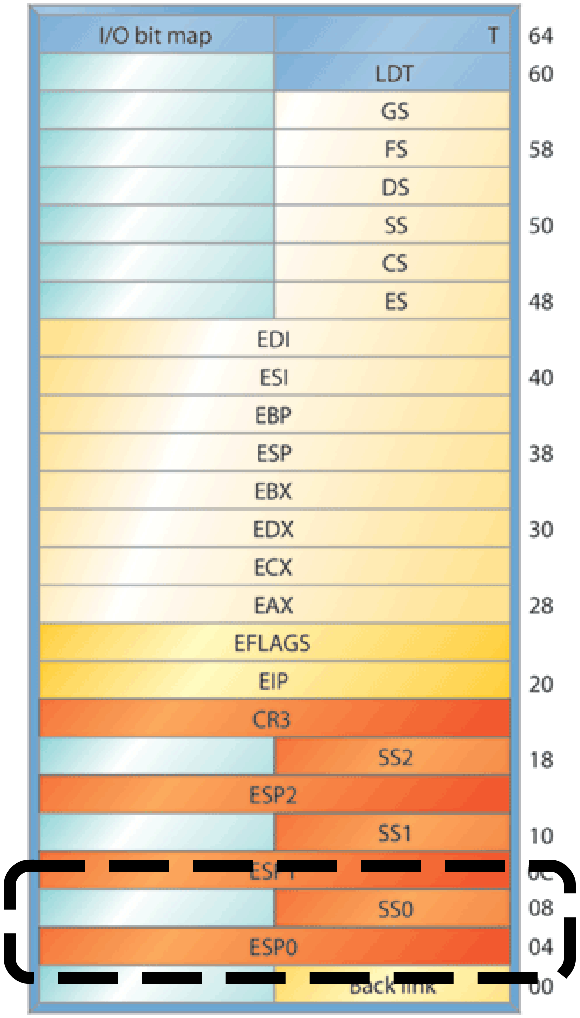
Executing in User space

- Switch stack to a kernel switch

How to switch stack?

Task State Segment

- Specialized segment for hardware support for multitasking
- TSS stored in memory
  - Pointer stored as part of GDT
  - Loaded by instruction : ltr(SEG_TSS << 3) in switchuvm()
- Important contents of TSS used to find the new stack
  - **SS0 :** the stack segment (in kernel)
  - **ESP0 :** stack pointer (in kernel)



ref : (switchuvm) ([18],1873), taskstate ([08],0850)

3 Saving Program State

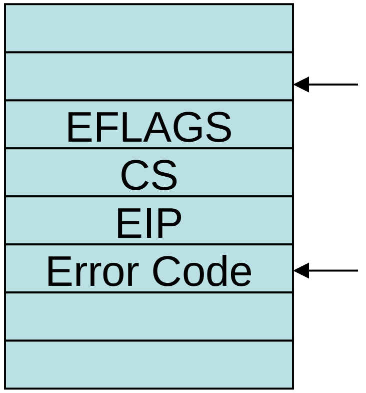
Why?

- Current program being executed must be able to resume after interrupt service is completed

3 Saving Program State

Done automatically by CPU

When no stack switch occurs
use existing stack



ESP before

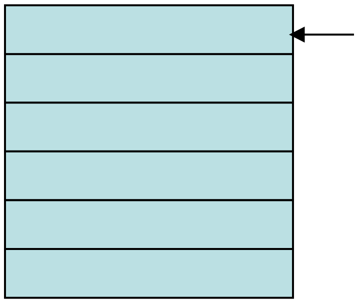
EFLAGS
CS
EIP
Error Code

ESP after

SS : No change
ESP : new frame pushed

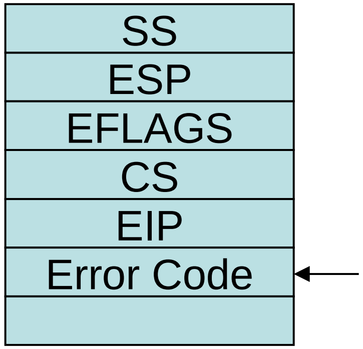
Error code is only for some exceptions. Contains additional Information.

When stack switch occurs
also save the previous SS and ESP



ESP before

Interrupted Procedure Stack (in user space)



SS
ESP
EFLAGS
CS
EIP
Error Code

Procedure's kernel stack

ESP after

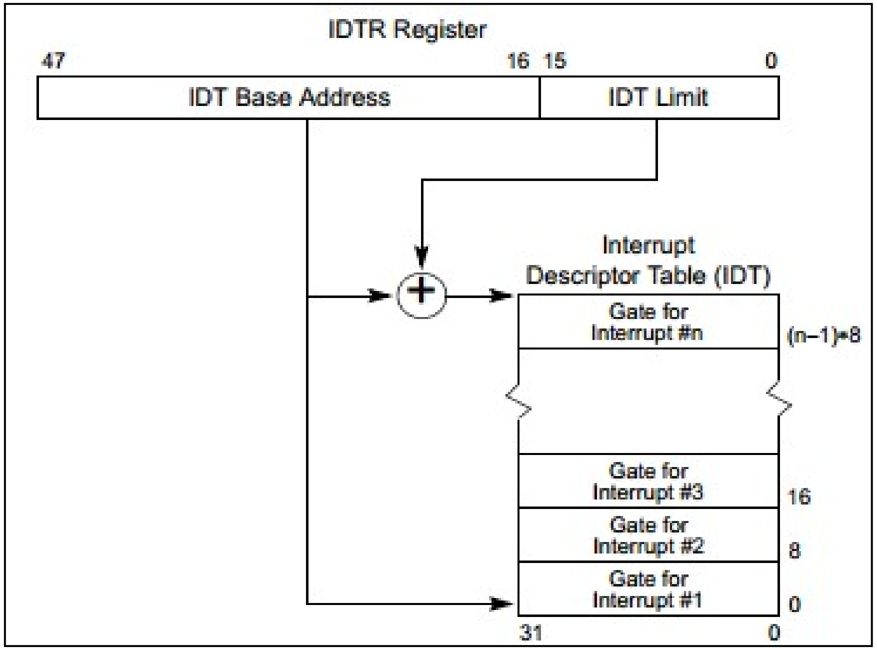
SS : from TSS (SS0)
ESP : from TSS (ESP0)

Finding the Interrupt/Exception Service Routine

4

- IDT : Interrupt descriptor table
  - Also called Interrupt vectors
  - Stored in memory and pointed to by IDTR
  - Conceptually similar to GDT and LDT
  - Initialized by OS at boot

Done automatically by CPU



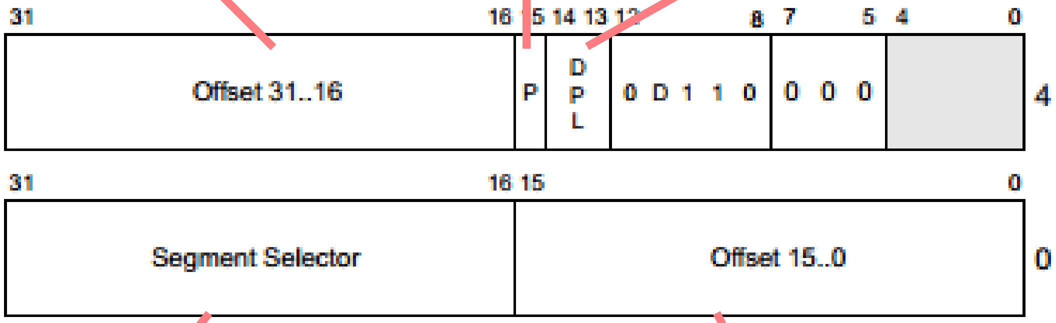
Selected Descriptor =
Base Address + (Vector * 8)

Interrupt Gate Descriptor

points to offset in the segment which contains the interrupt handler (higher order bits)

1 Segment present
0 Segment absent

privilege level



points to a segment descriptor for executable code in the GDT

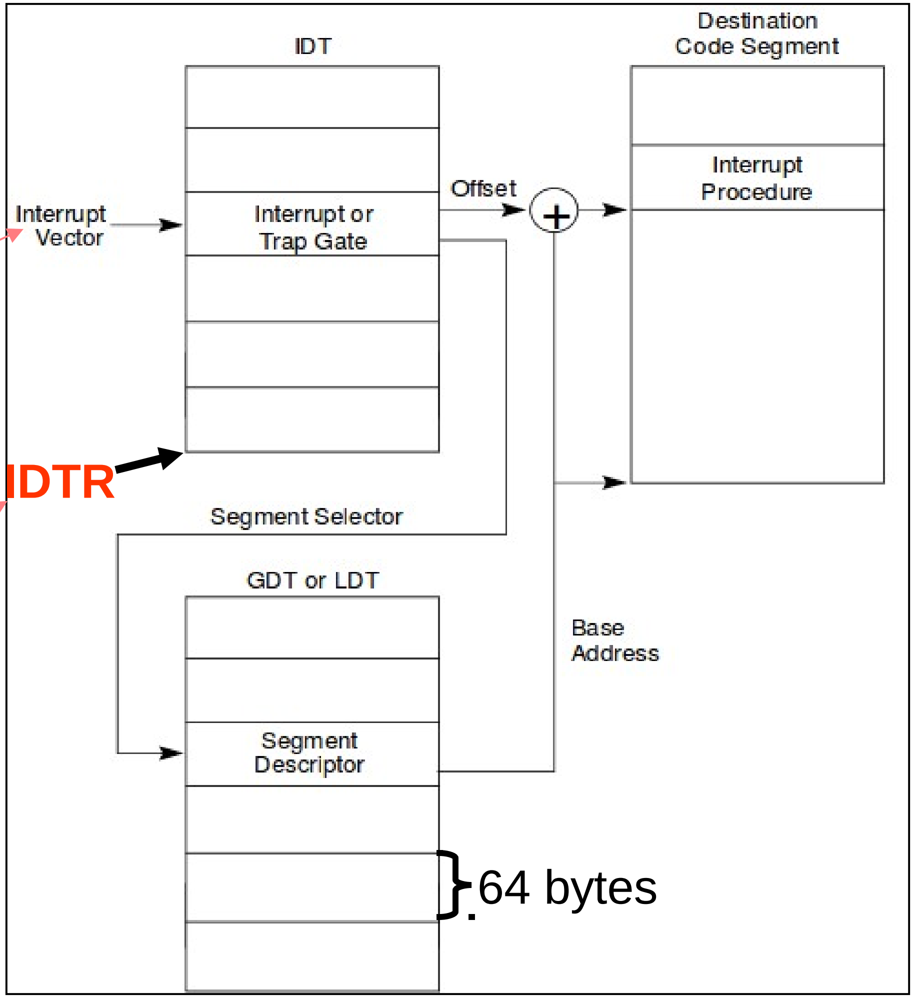
points to offset in the segment which contains the interrupt handler (lower order bits)

ref : SETGATE (0921), gatedesc (0901)

Getting to the Interrupt Procedure

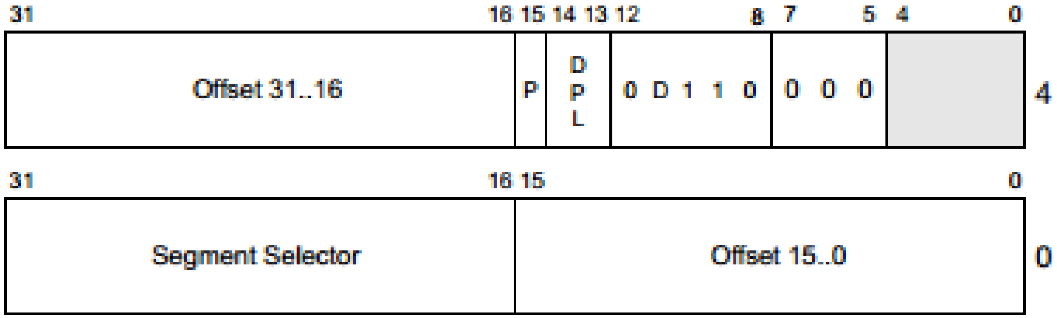
(obtained from either the PIC or APIC)

IDTR : pointer to IDT table in memory



Done automatically by CPU

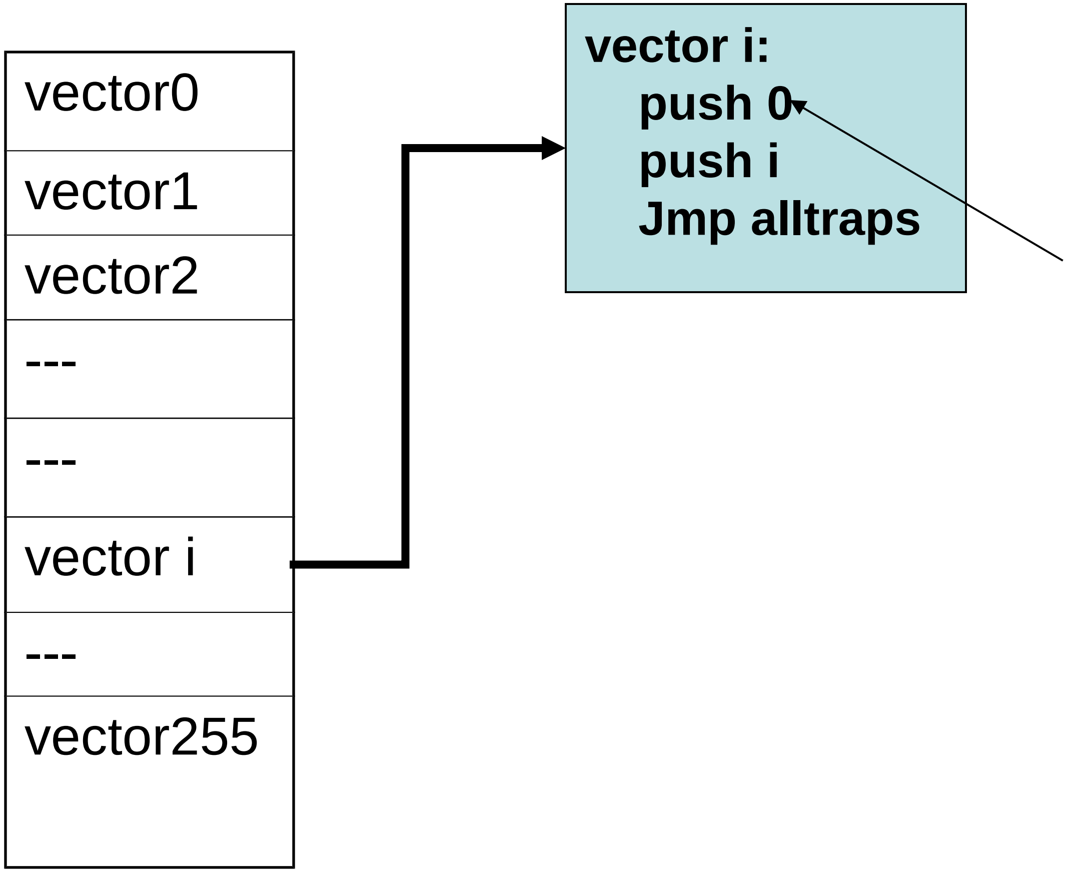
Setting up IDT in xv6



- Array of 256 gate descriptors (idt)
- Each idt has
  - Segment Selector : SEG_KCODE
    - This is the offset in the GDT for kernel code segment
  - Offset : (interrupt) vectors (generated by Script vectors.pl)
    - Memory addresses for interrupt handler
    - 256 interrupt handlers possible
- Load IDTR by instruction lidt
  - The IDT table is the same for all processors.
  - For each processor, we need to explicetly load lidt (idtinit())

ref : tvinit() (3317) and idtinit() in trap.c

Interrupt Vectors in xv6



vector0

vector1

vector2

---

vector i

---

vector255

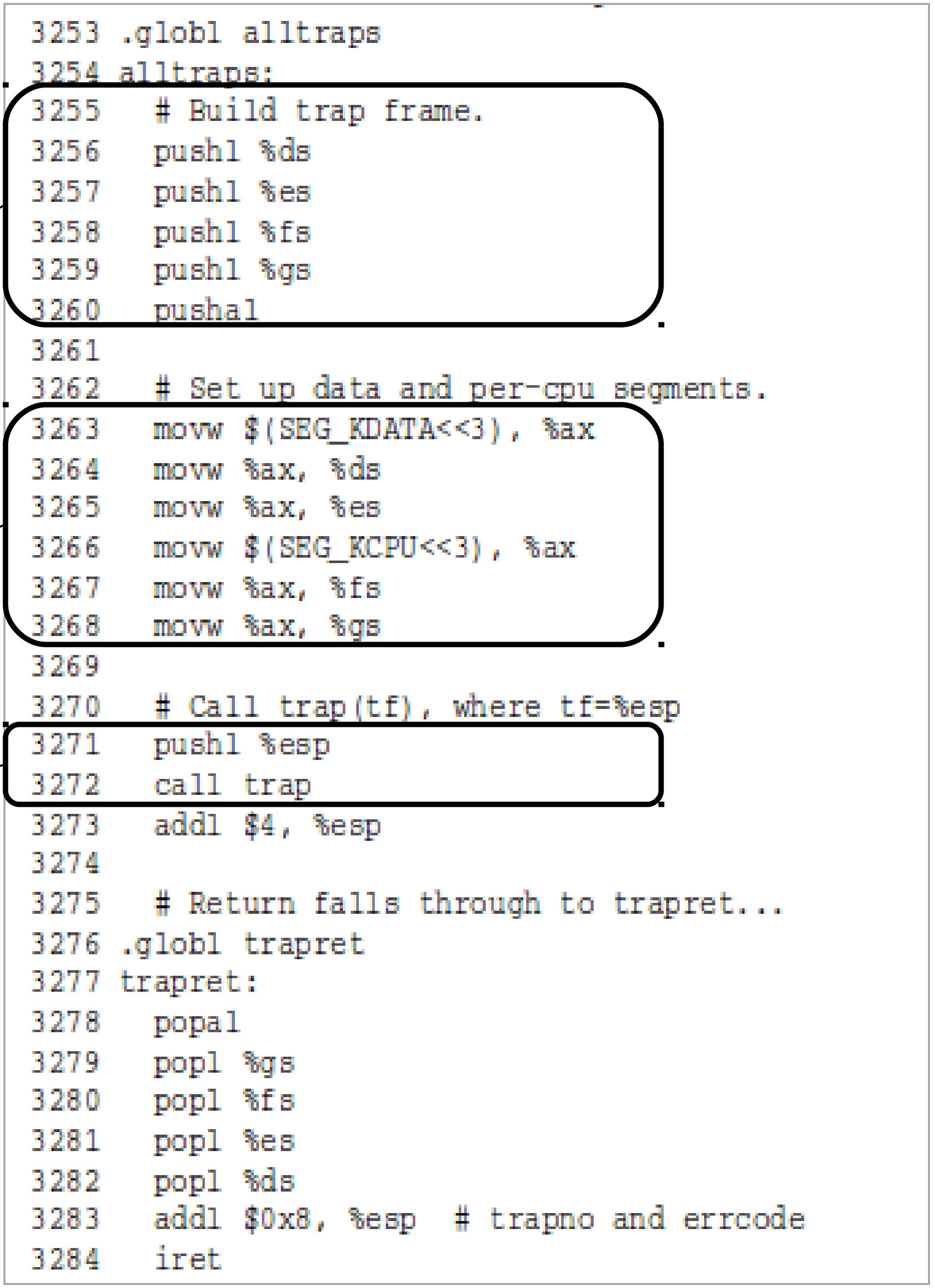
```
vector i:
    push 0
    push i
    Jmp alltraps
```

Error code:
Hardware pushes error
Code for some exceptions.
For others, xv6 pushes 0.

ref : vectors.s [generated by vectors.pl (run *$perl vectors.pl*)] ([32])

5

alltraps



```
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Creates a trapframe

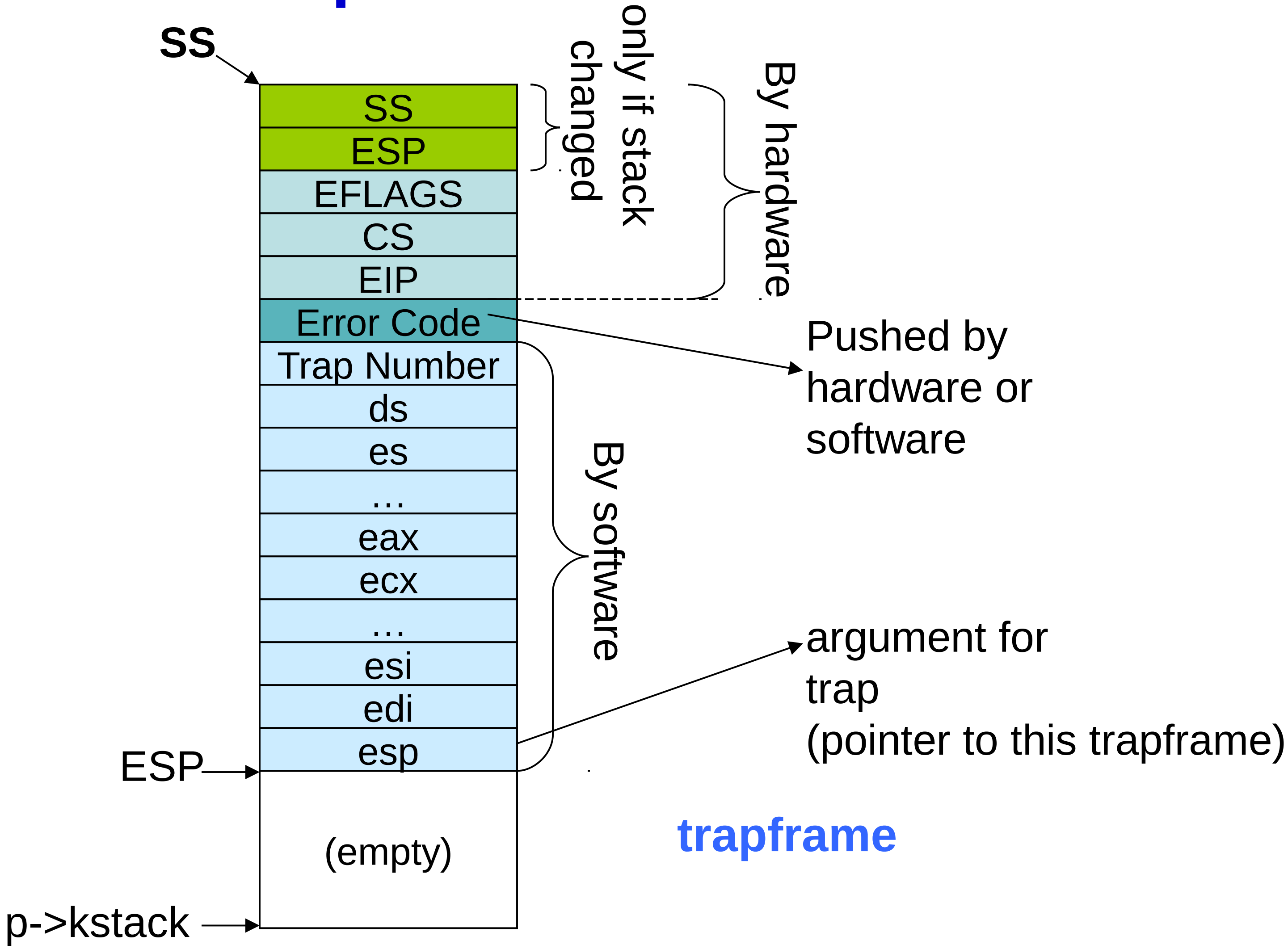
Stack frame used for interrupt

Setup kernel data and code segments

Invokes trap (3350 [33])

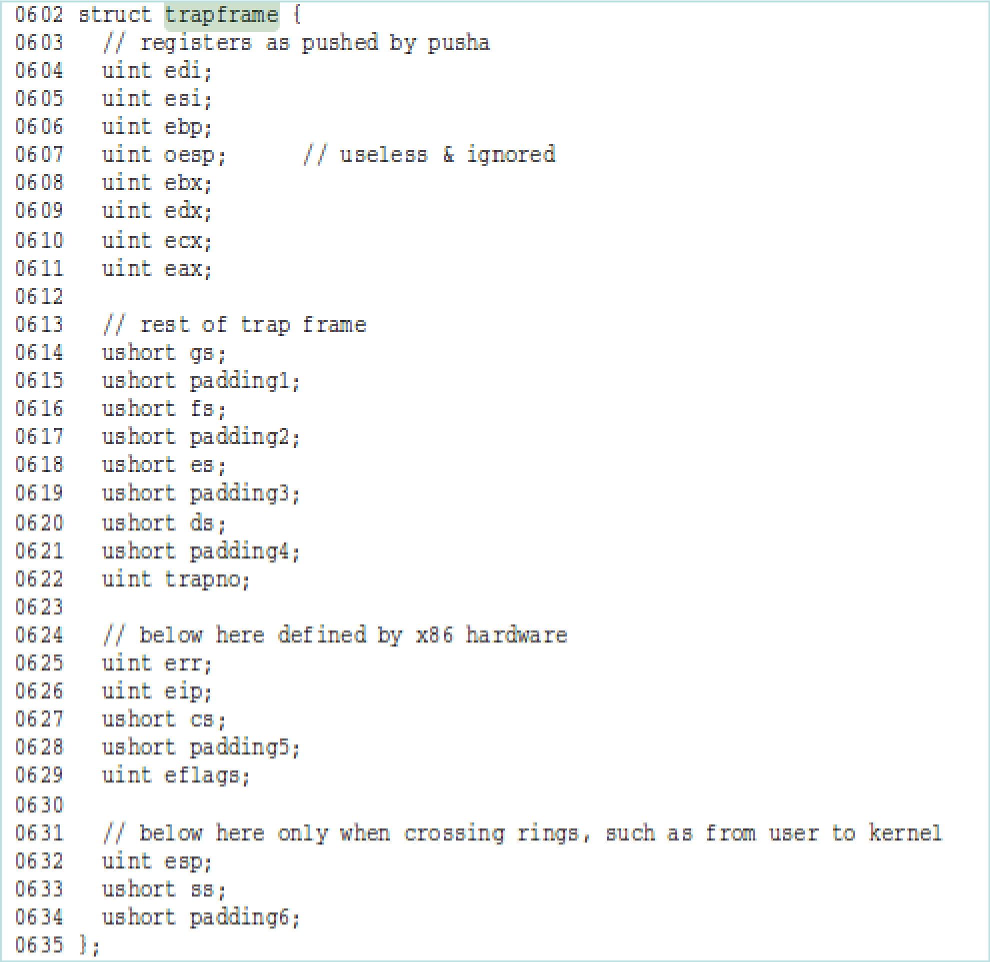
ref : trapasm.S [32] (alltraps), trap.c [33] (trap())

trapframe



ref : struct trapframe in x86.h (0602 [06])

trapframe struct

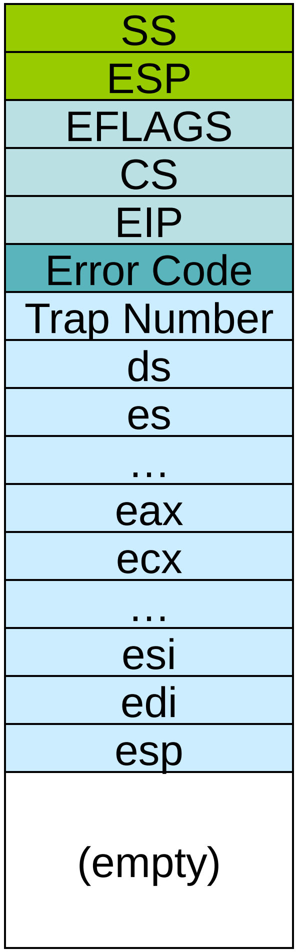


```
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```



SS

ESP

EFLAGS

CS

EIP

Error Code

Trap Number

ds

es

…

eax

ecx

…

esi

edi

esp

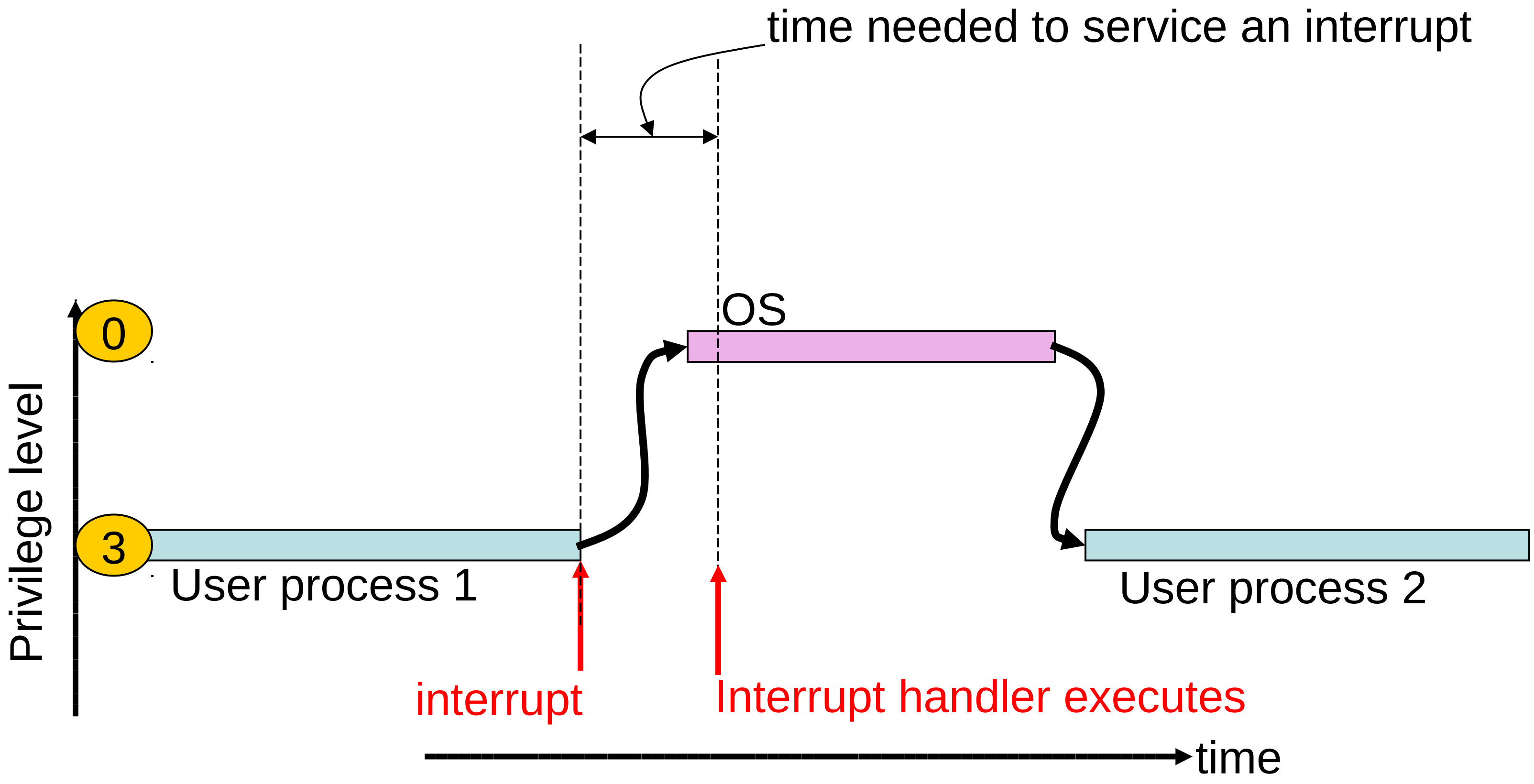
(empty)

4

Interrupt Handlers

- Typical Interrupt Handler
  - Save additional CPU context (written in assembly) (done by alltraps in xv6)
  - Process interrupt (communicate with I/O devices)
  - Invoke kernel scheduler
  - Restore CPU context and return (written in assembly)

Interrupt Latency


time needed to service an interrupt
OS
0
3
Privilege level
User process 1
User process 2
interrupt
Interrupt handler executes
time

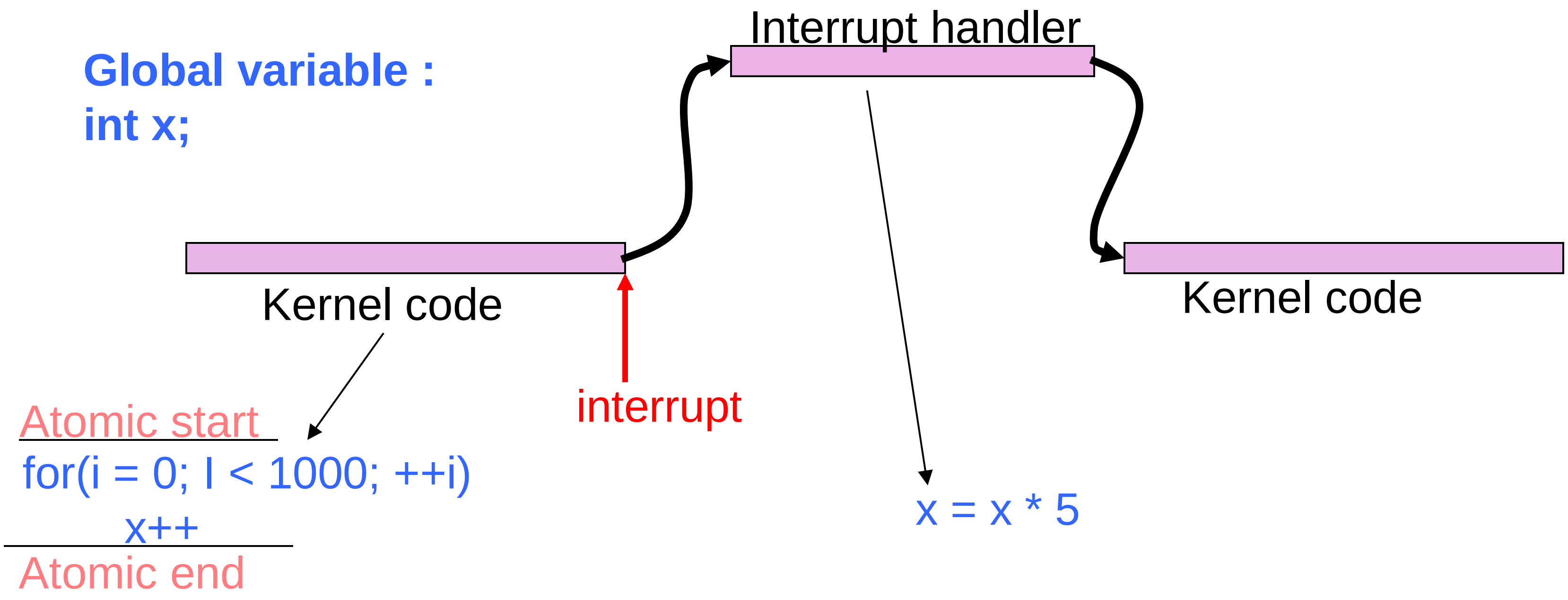
Interrupt latency can be significant

Importance of Interrupt Latency

- Real time systems
  - OS should ‘guarantee’ interrupt latency is less than a specified value
- Minimum Interrupt Latency
  - Mostly due to the interrupt controller
- Maximum Interrupt Latency
  - Due to the OS
  - Occurs when interrupt handler cannot be serviced immediately
    - Eg. when OS executing atomic operations, interrupt handler would need to wait till completion of atomic operations.

Atomic Operations

Global variable :
int x;



Interrupt handler

Kernel code

Kernel code

interrupt

Atomic start

```
for(i = 0; I < 1000; ++i)
        x++
```

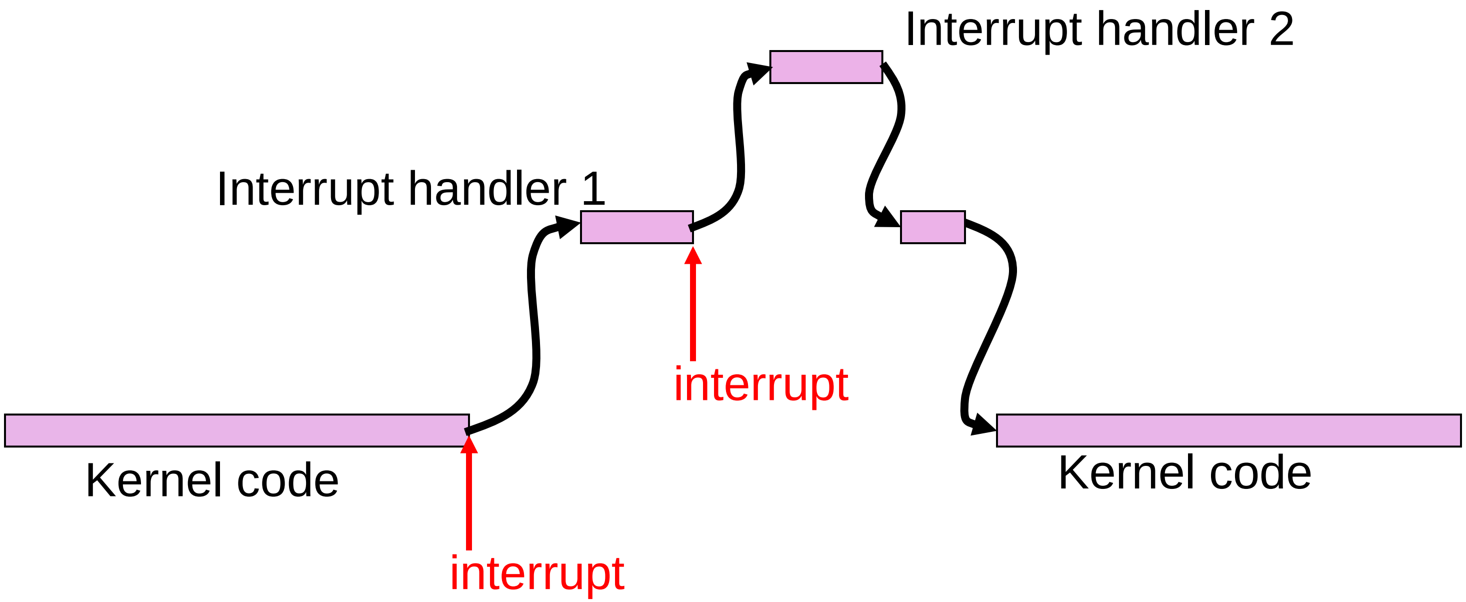
Atomic end

x = x * 5

Value of x depends on whether an interrupt occurred or not!

Solution : make the part of code atomic (i.e. disable interrupts while executing this code)

Nested Interrupts



- Typically interrupts disabled until handler executes
  - This reduces system responsiveness
- To improve responsiveness, enable Interrupts within handlers
  - This often causes nested interrupts
  - Makes system more responsive but difficult to develop and validate
- **Interrupt handler approach:** design interrupt handlers to be small so that nested interrupts are less likely

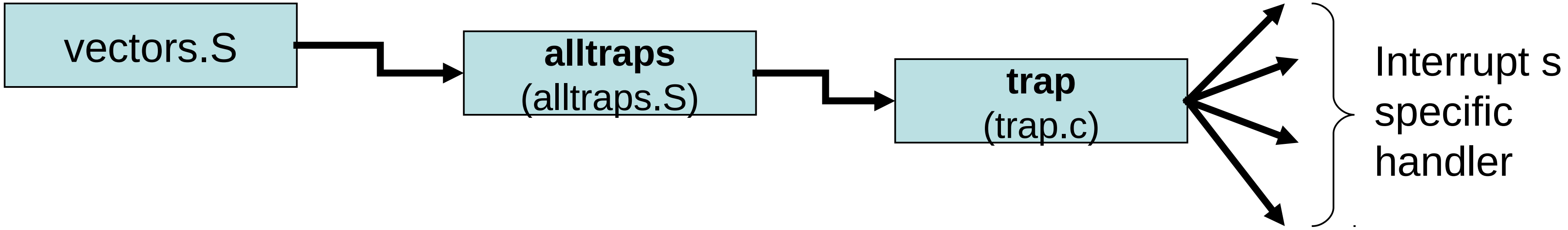
Small Interrupt Handlers

- Do as little as possible in the interrupt handler
  - Often just queue a work item or set a flag
- Defer non-critical actions till later

Top and Bottom Half Technique (Linux)

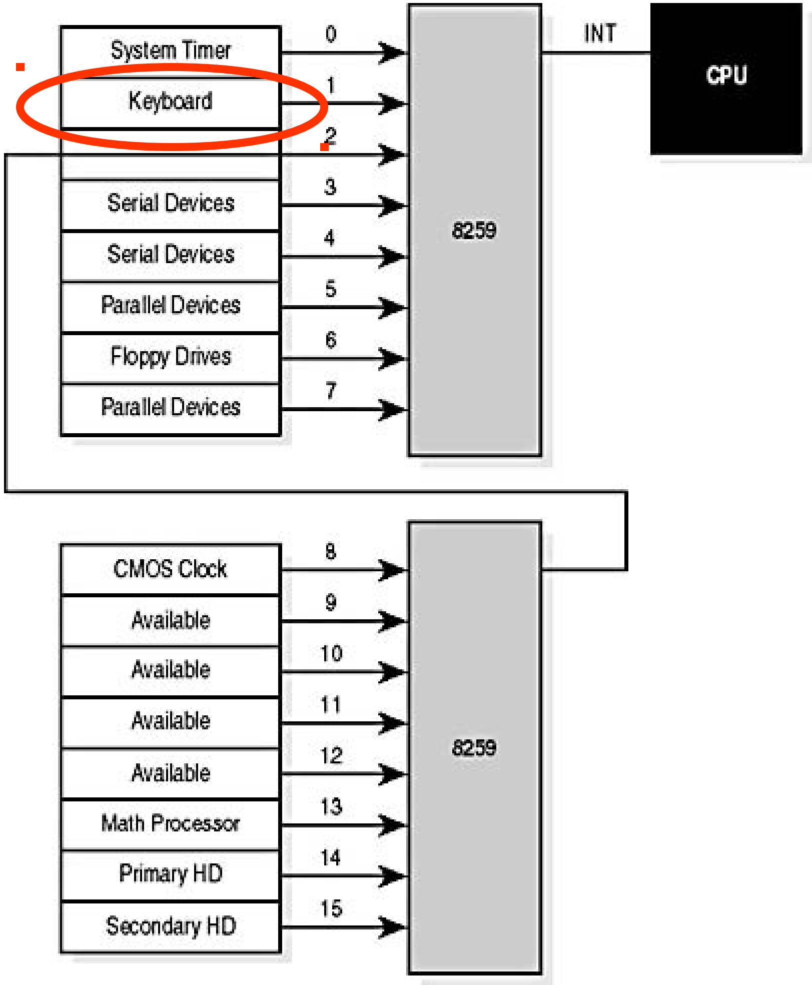
- Top half : do minimum work and return from interrupt handler
  - Saving registers
  - Unmasking other interrupts
  - Restore registers and return to previous context
- Bottom half : deferred processing
  - eg. Workqueue
  - Can be interrupted

Interrupt Handlers in xv6



Example (Keyboard Interrupt in xv6)

- Keyboard connected to second interrupt line in 8259 master
- Mapped to vector 33 in xv6 (T_IRQ0 + IRQ_KBD).
- In function trap, invoke keyboard interrupt (kbdintr), which is redirected to consleintr



Keyboard Interrupt Handler

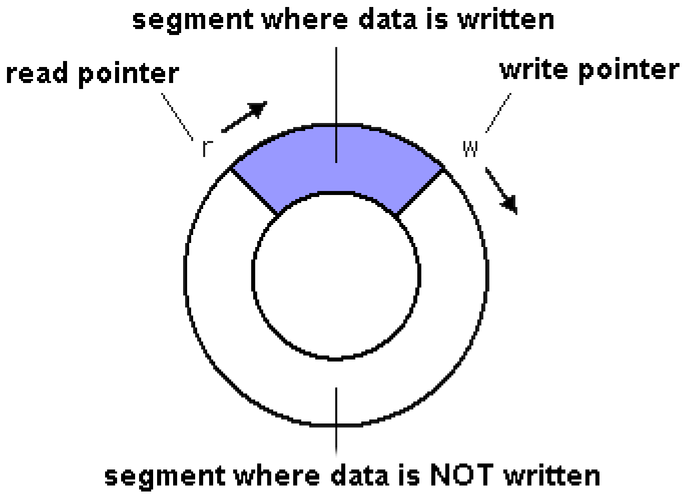
consoleintr (console.c)

get pressed character (kbdgetc (kbd.c0)

talks to keyboard through specific predifined io ports

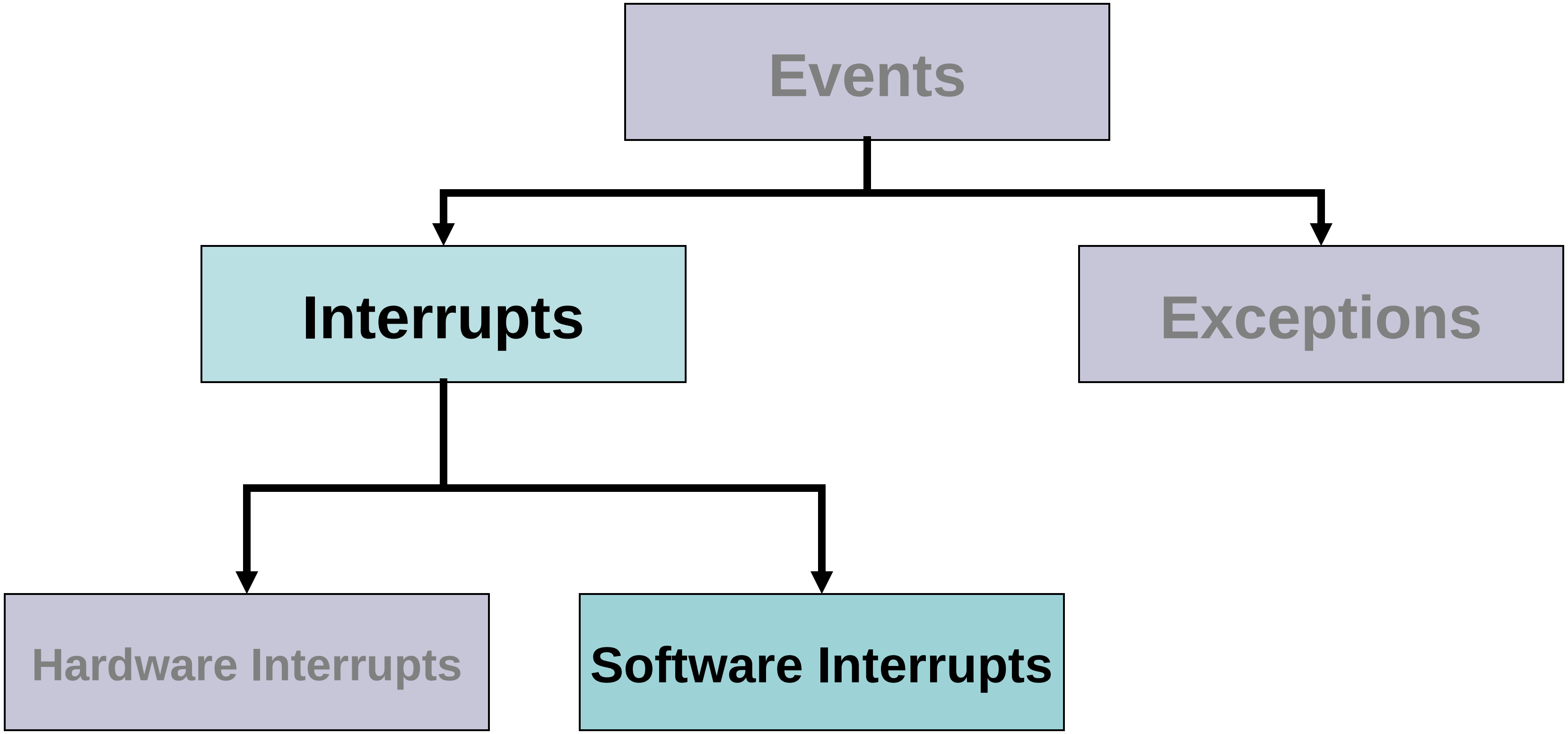
Service special characters

Push into circular buffer


segment where data is written
read pointer
write pointer
r
w
segment where data is NOT written

System Calls and Exceptions

Events



Events

Interrupts

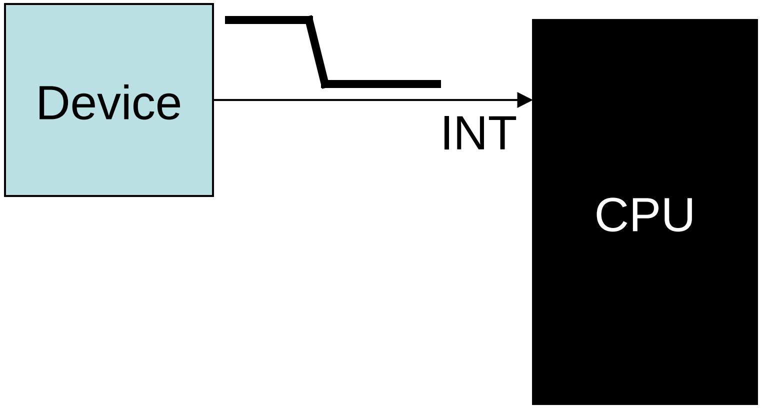
Exceptions

Hardware Interrupts

Software Interrupts

Hardware vs Software Interrupt

Hardware Interrupt



- A device (like the PIC) asserts a pin in the CPU

Software Interrupt

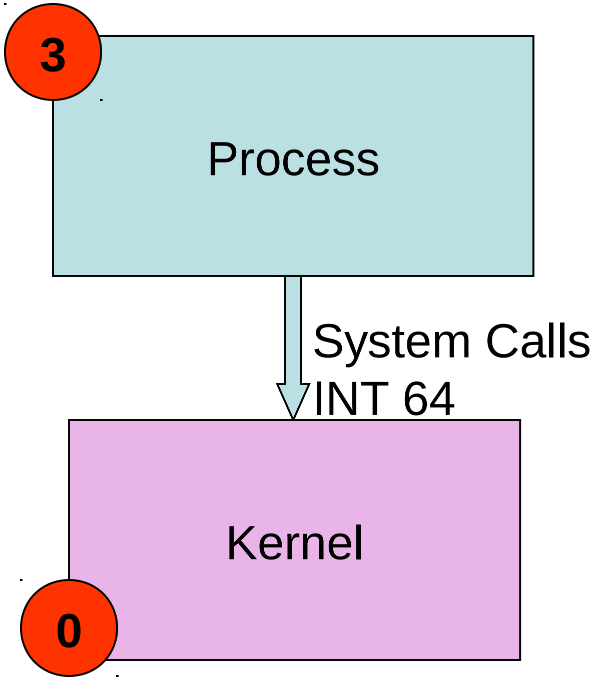
```
.
.
INT x
.
.
```

- An instruction which when executed causes an interrupt

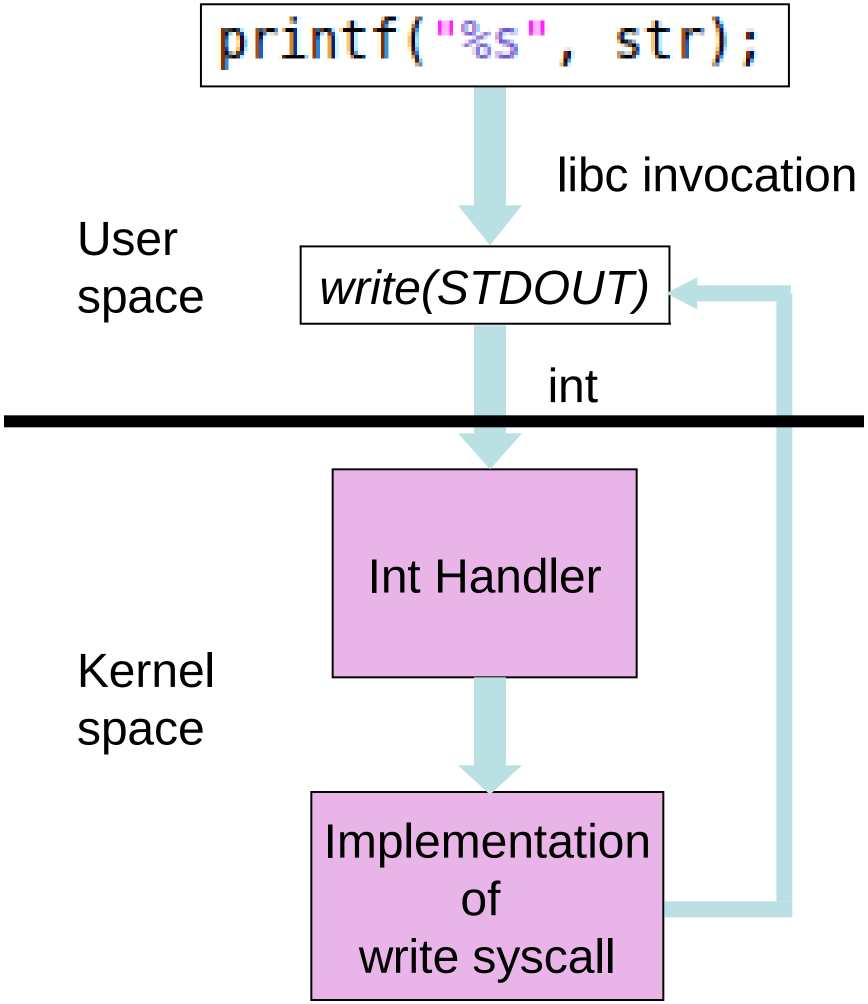
Software Interrupt

Software interrupt used for implementing system calls

- In Linux INT 128, is used for system calls
- In xv6, INT 64 is used for system calls



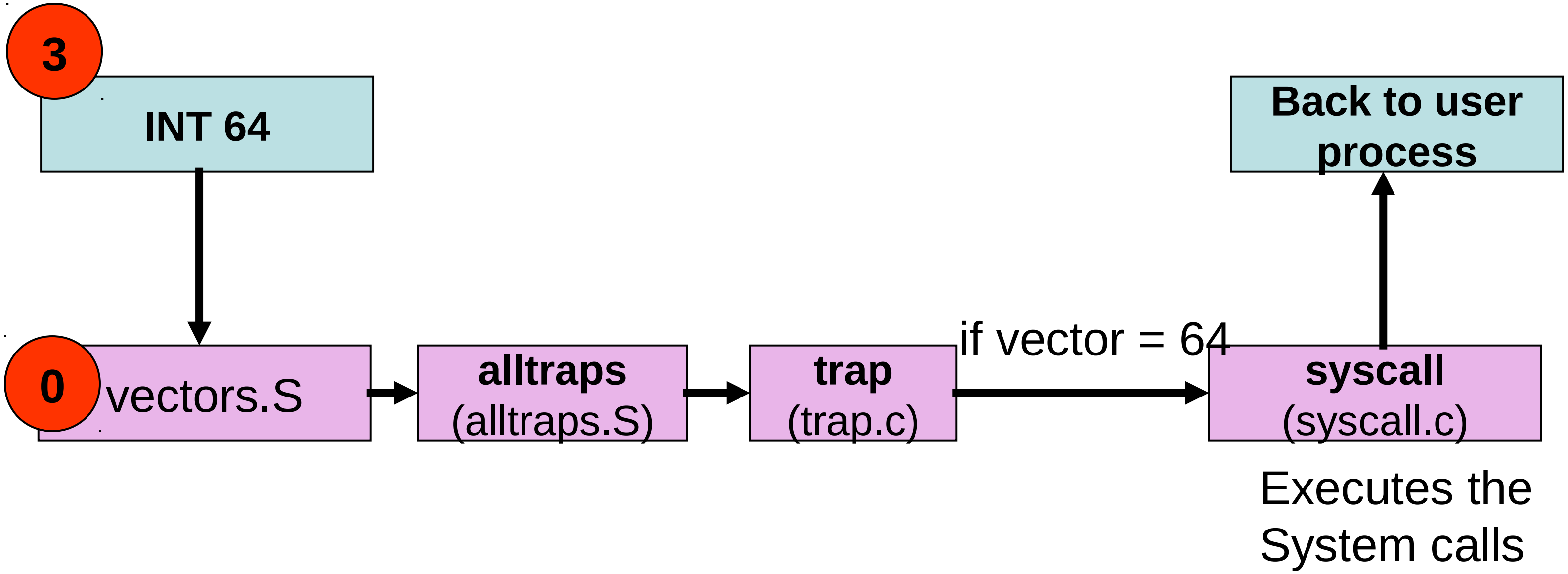
Example (write system call)



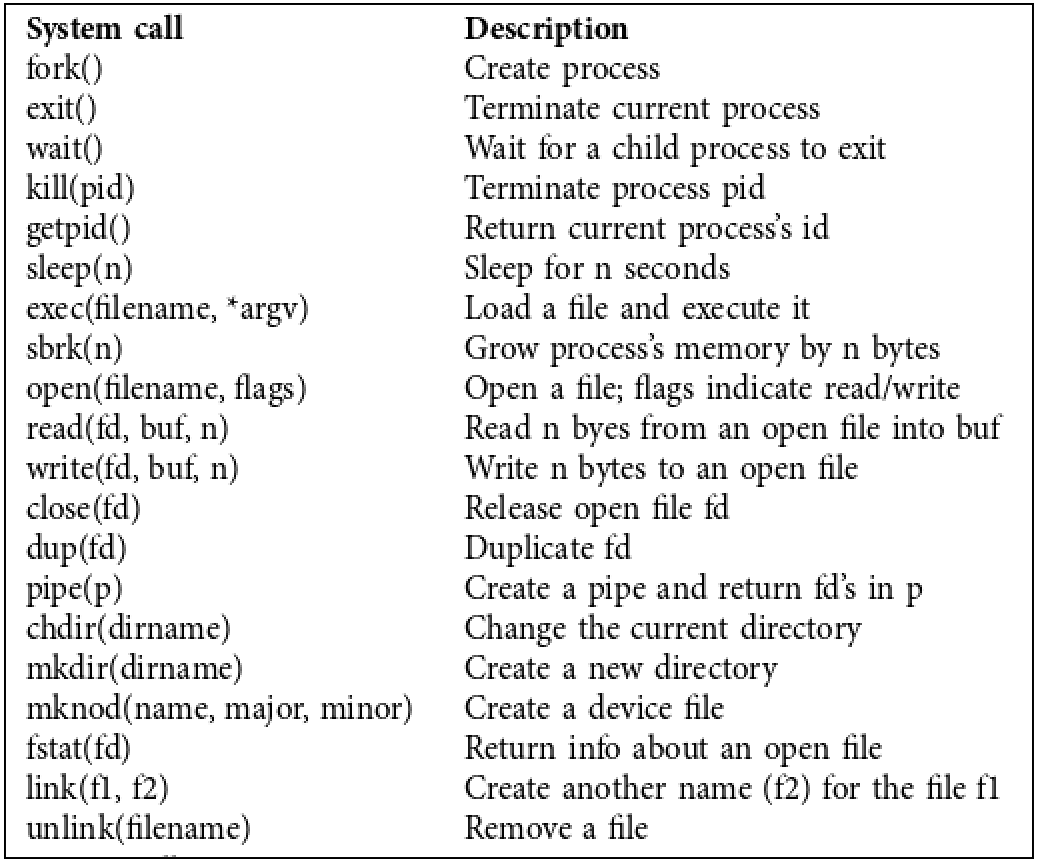
```
printf("%s", str);
```

System call processing in kernel

Almost similar to hardware interrupts



System Calls in xv6



System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; flags indicate read/write
read(fd, buf, n)	Read n byes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

How does the OS distinguish between the system calls?

System Call Number

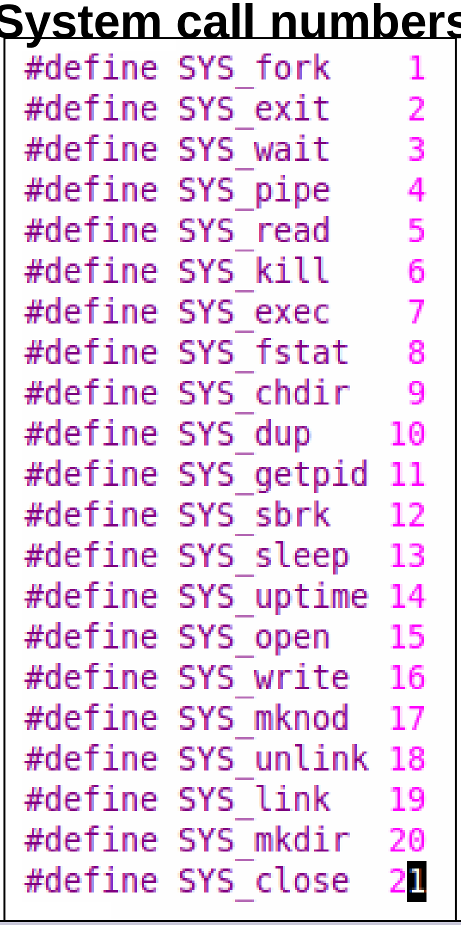
System call number used to distinguish between system calls

System call number

```
mov x, %eax
INT 64
```

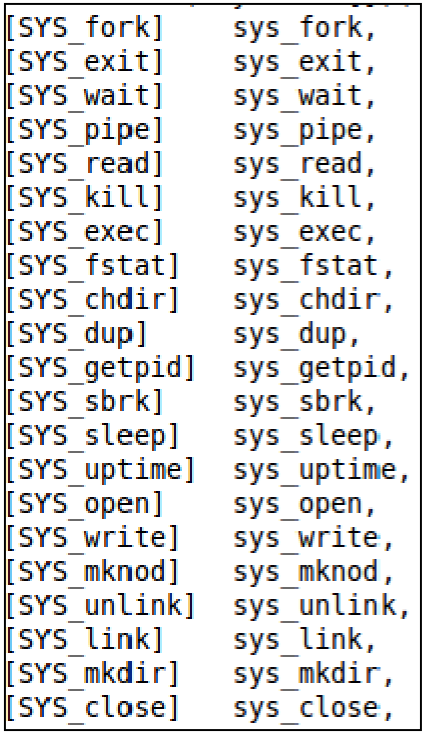
Based on the system call number function syscall invokes the corresponding syscall handler

System call numbers



```
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

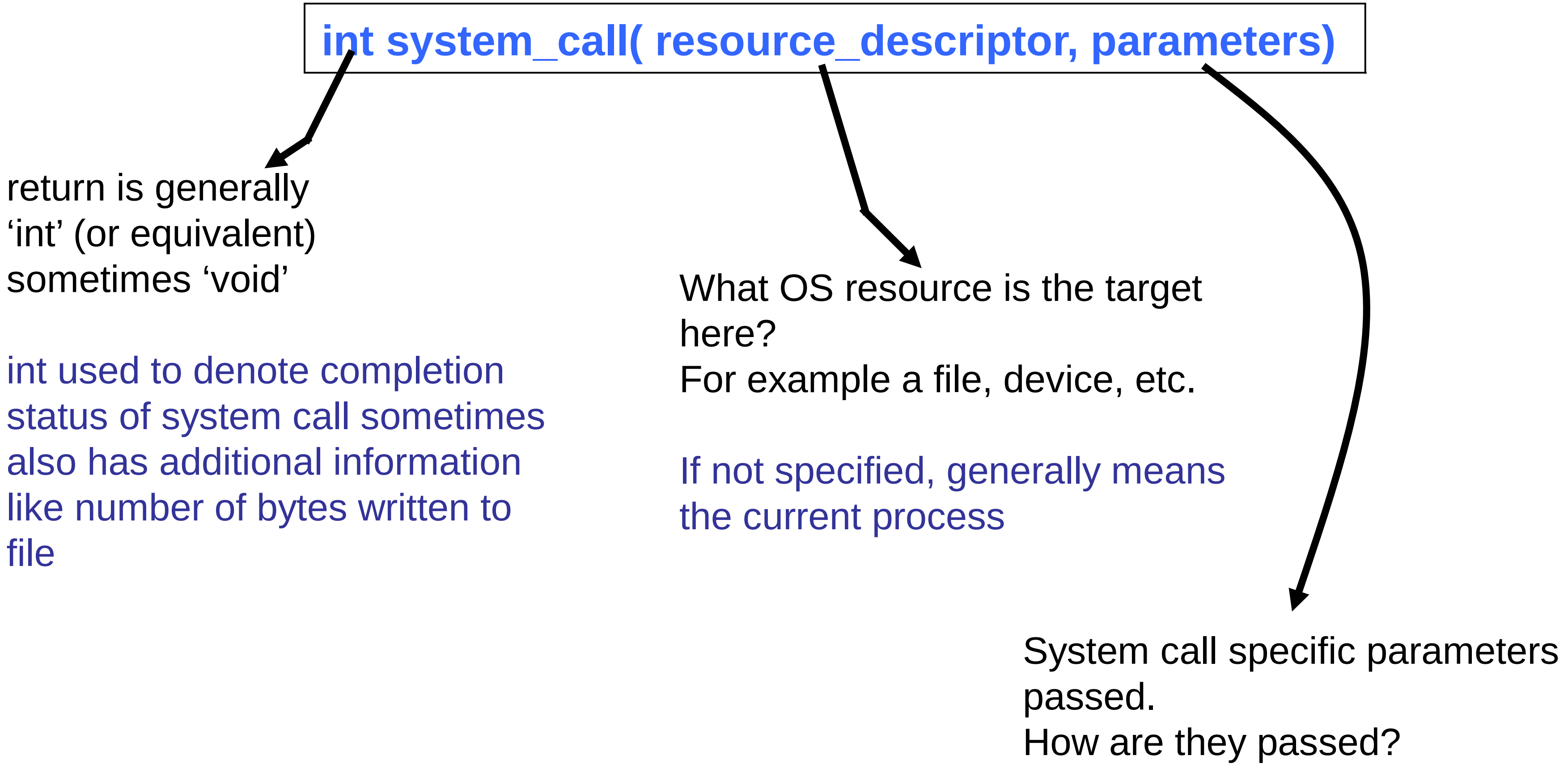
System call handlers



```
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
```

ref : syscall.h, syscall() in syscall.c

Prototype of a typical System Call



int system_call(resource_descriptor, parameters)

return is generally
‘int’ (or equivalent)
sometimes ‘void’

int used to denote completion status of system call sometimes also has additional information like number of bytes written to file

What OS resource is the target here?
For example a file, device, etc.

If not specified, generally means the current process

System call specific parameters passed.
How are they passed?

Passing Parameters in System Calls

- Passing parameters to system calls not similar to passing parameters in function calls
  - Recall stack changes from user mode stack to kernel stack.
- Typical Methods
  - Pass by Registers (eg. Linux)
  - Pass via user mode stack (eg. xv6)
    - Complex
  - Pass via a designated memory region
    - Address passed through registers

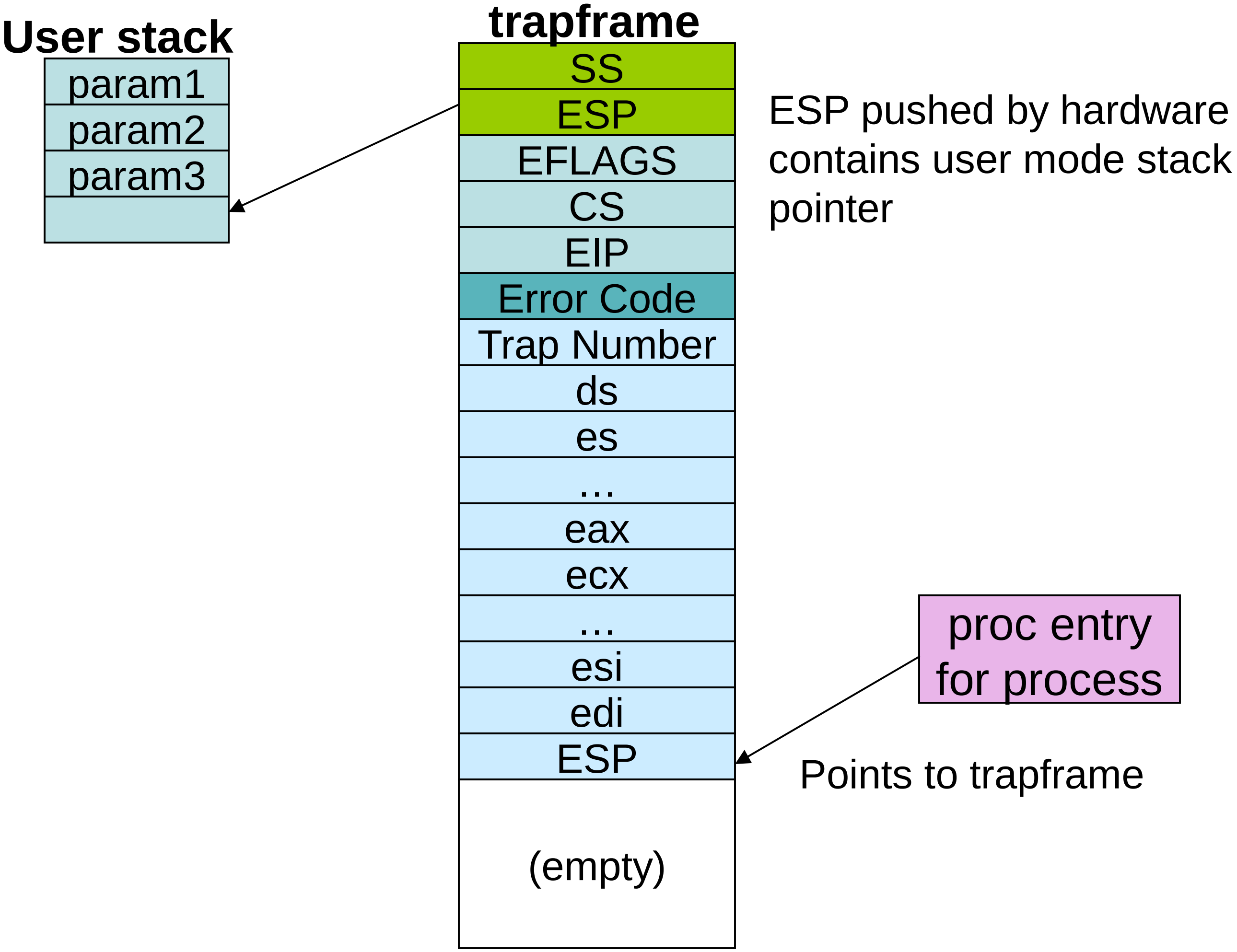
Pass By Registers (Linux)

- System calls with fewer than 6 parameters passed in registers
  - %eax (sys call number), %ebx, %ecx,, %esi, %edi, %ebp
- If 6 or more arguments
  - Pass pointer to block structure containing argument list
- Max size of argument is the register size (eg. 32 bit)
  - Larger pointers passed through pointers

Pass via User Mode Stack (xv6)

User process

```
push param1
push param2
push param3
mov sysnum, %eax
int 64
```



User stack

param1

param2

param3

trapframe

SS

ESP

EFLAGS

CS

EIP

Error Code

Trap Number

ds

es

…

eax

ecx

…

esi

edi

ESP

(empty)

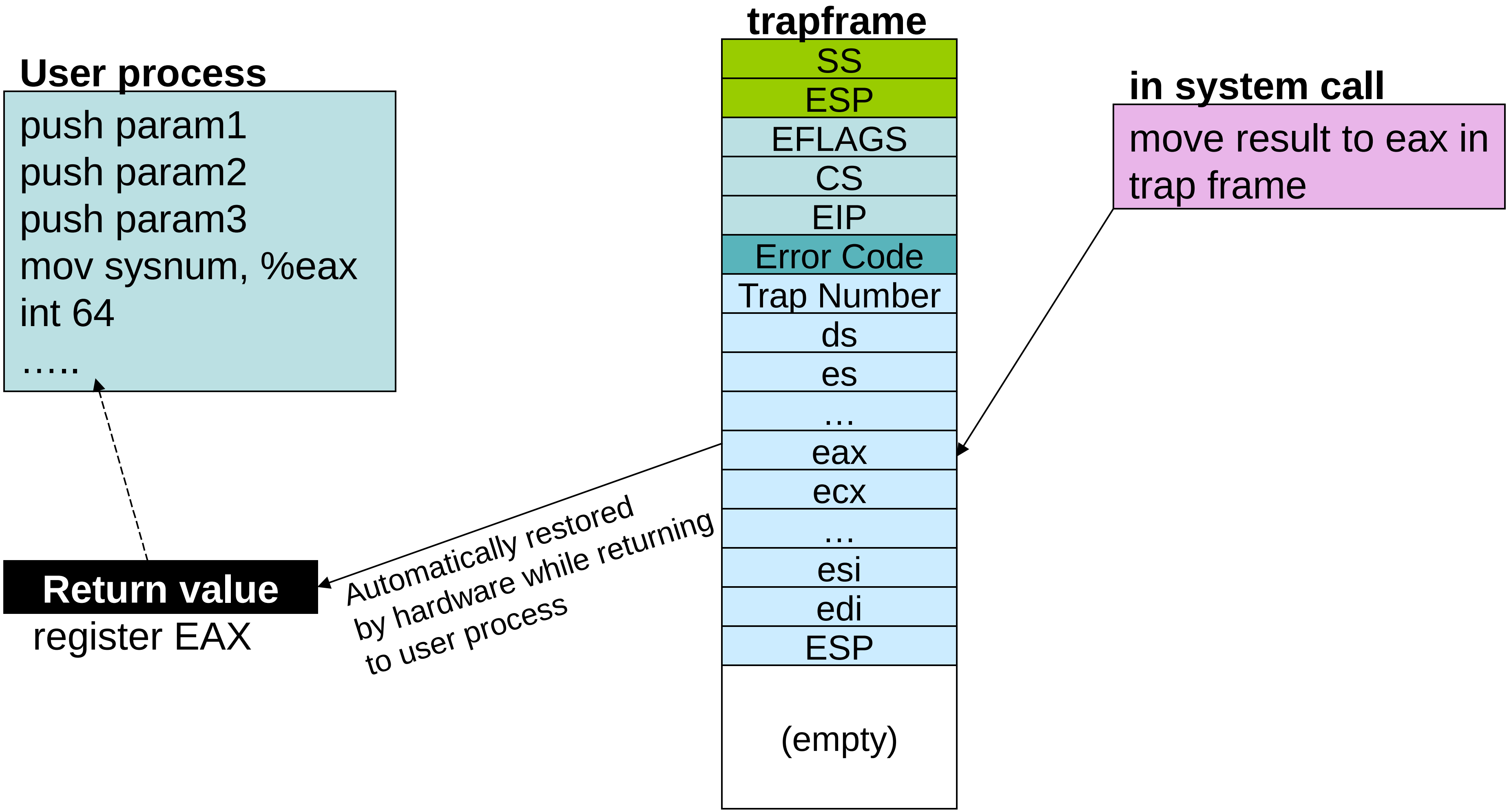
ESP pushed by hardware contains user mode stack pointer

proc entry for process

Points to trapframe

ref : sys_open (sysfile.c), argint, fetchint (syscall.c)

Returns from System Calls



User process

```
push param1
push param2
push param3
mov sysnum, %eax
int 64
…..
```

trapframe

SS
ESP
EFLAGS
CS
EIP
Error Code
Trap Number
ds
es
…
eax
ecx
…
esi
edi
ESP
(empty)

in system call

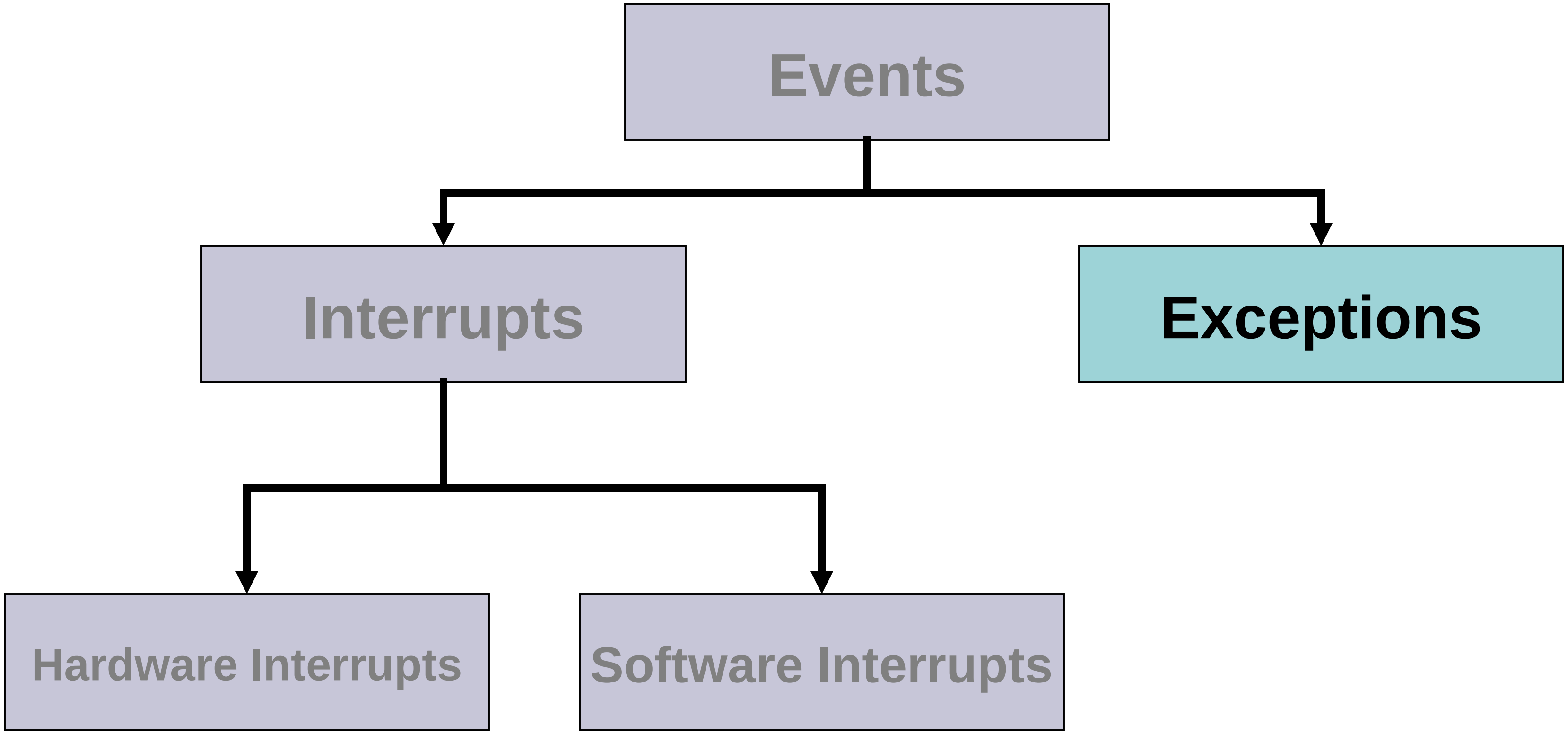
move result to eax in trap frame

Return value

register EAX

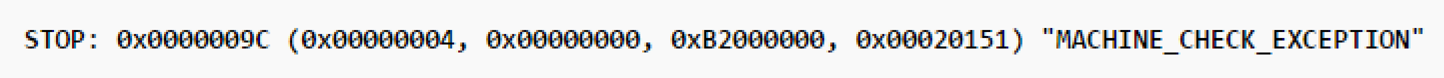
Automatically restored by hardware while returning to user process

Events


Events
Interrupts
Exceptions
Hardware Interrupts
Software Interrupts

Exception Sources

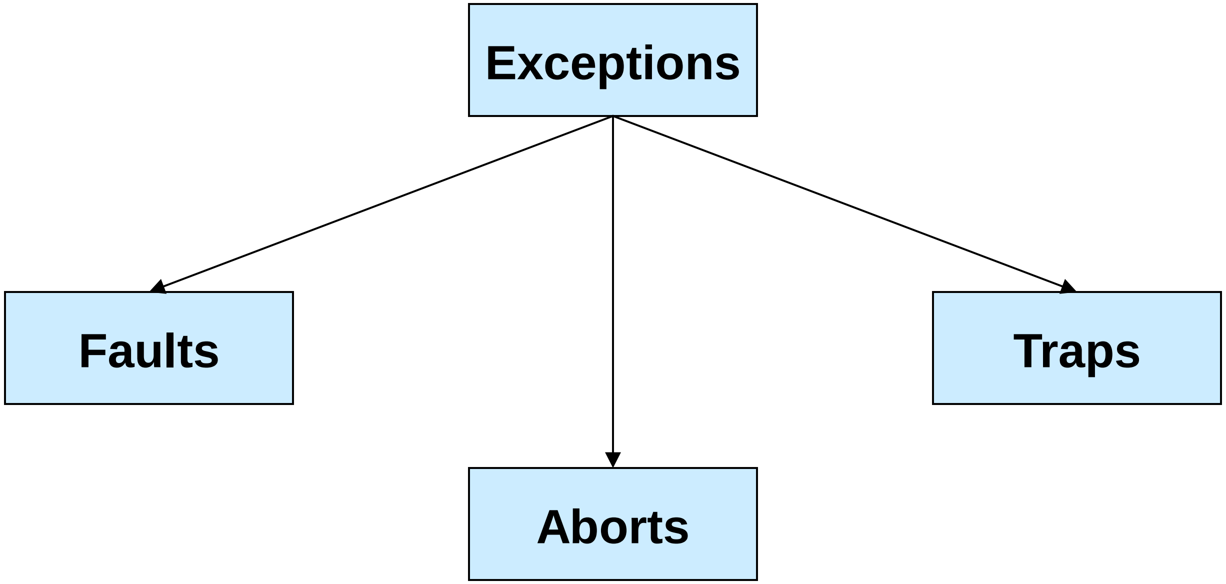
- Program-Error Exceptions
  - Eg. divide by zero
- Software Generated Exceptions
  - Example INTO, INT 3, BOUND
  - INT 3 is a break point exception
  - INTO overflow instruction
  - BOUND, Bound range exceeded
- Machine-Check Exceptions
  - Exception occurring due to a hardware error (eg. System bus error, parity errors in memory, cache memory errors)



```
STOP: 0x0000009C (0x00000004, 0x00000000, 0xB2000000, 0x00020151) "MACHINE_CHECK_EXCEPTION"
```

Microsoft Windows : Machine check exception

Exception Types



- Exceptions in the user space vs kernel space

Faults

Exception that generally can be corrected.

Once corrected, the program can continue execution.

Examples :

Divide by zero error

Invalid Opcode

Device not available

Segment not present

Page not present

Traps

Traps are reported immediately after the execution of the trapping instruction.

Examples:

Breakpoint

Overflow

Debug instructions

Aborts

Severe unrecoverable errors

Examples

Double fault : occurs when an exception is unhandled or when an exception occurs while the CPU is trying to call an exception handler.

Machine Check : internal errors in hardware detected. Such as bad memory, bus errors, cache errors, etc.