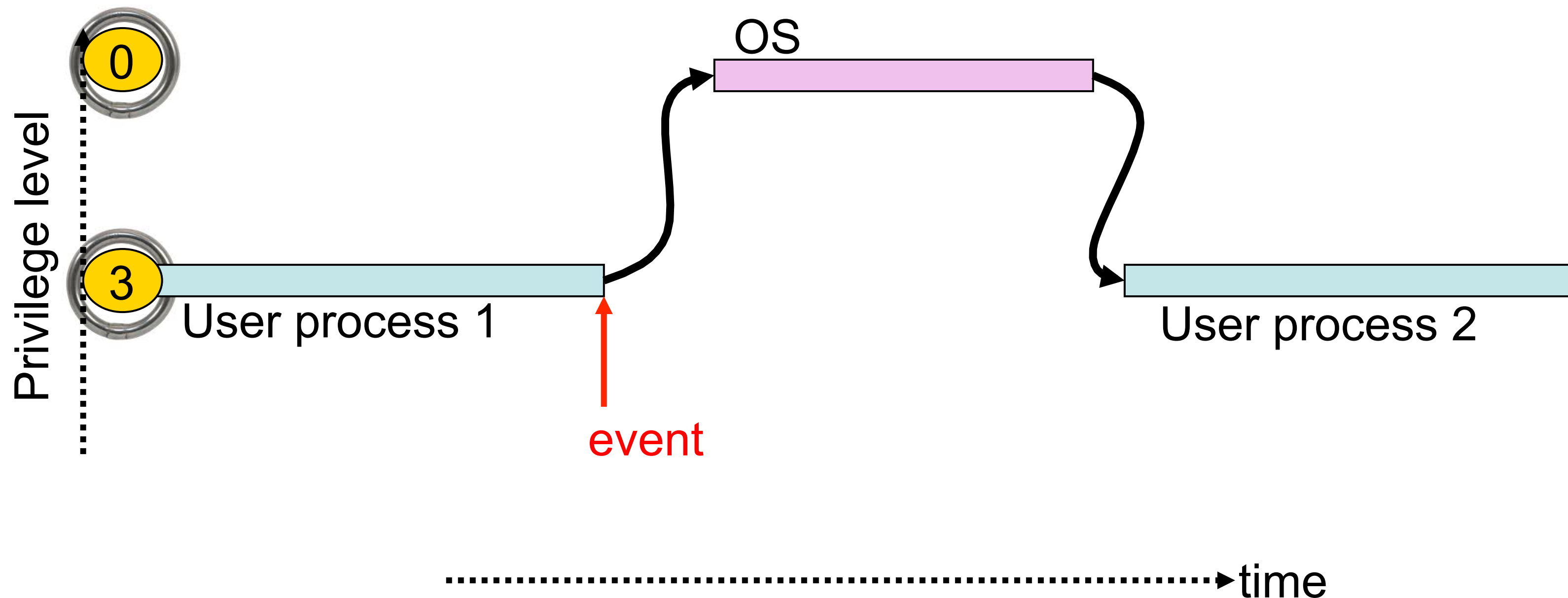
Interrupts, Exceptions, and System Calls

Chester Rebeiro

IIT Madras

OS & Events

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

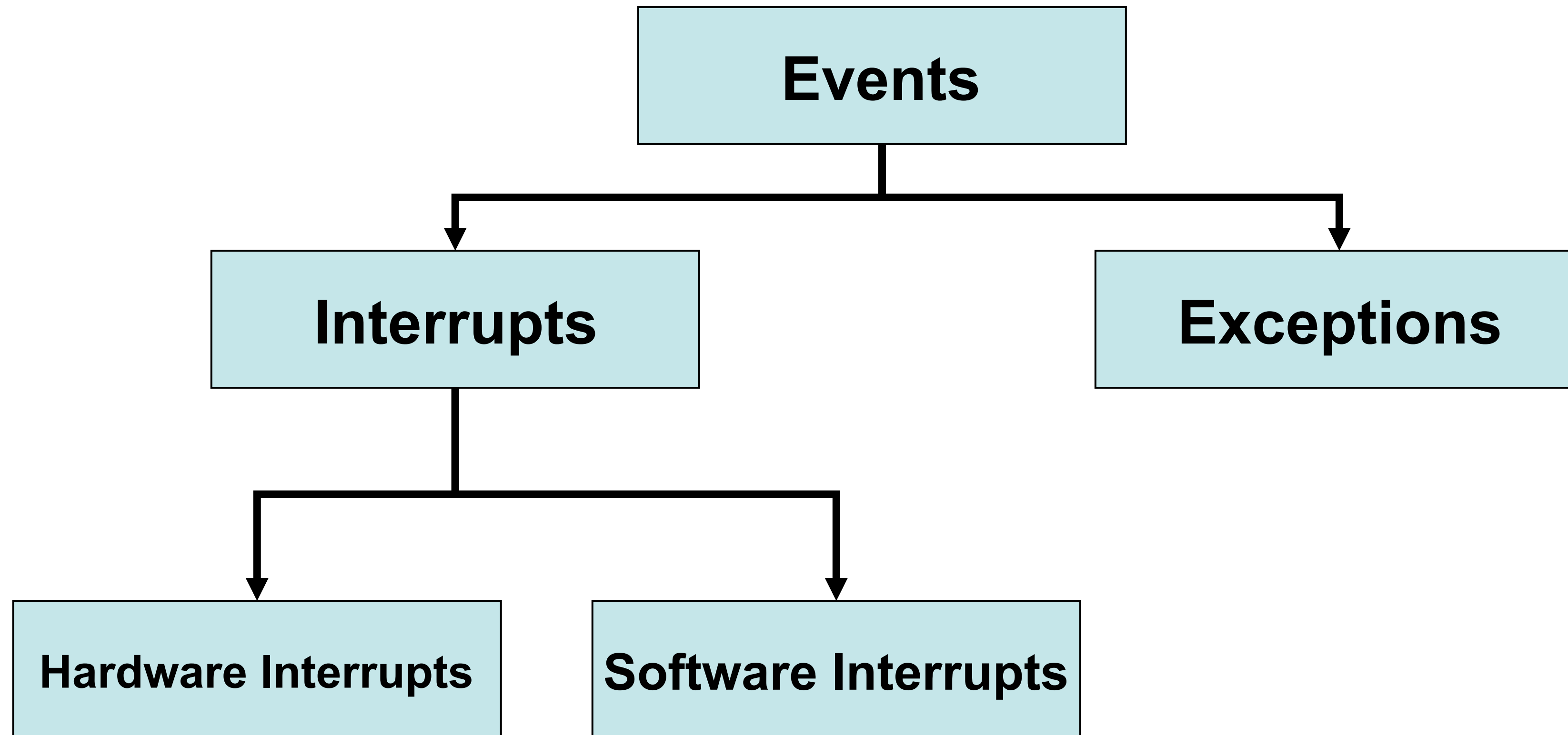


Why event driven design?

- 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

CR

Event Types



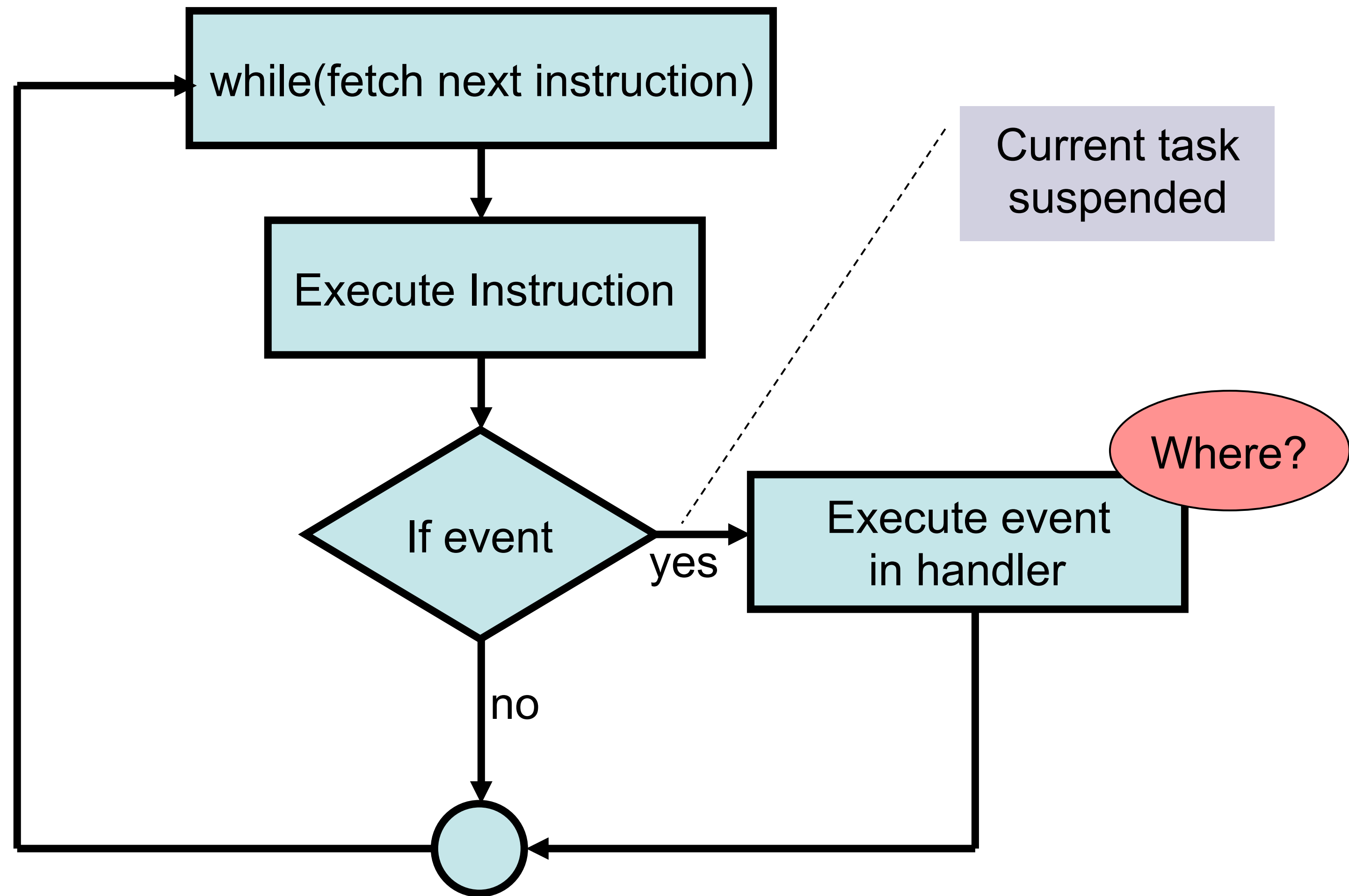
Events

- 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

CR

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

CR

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.

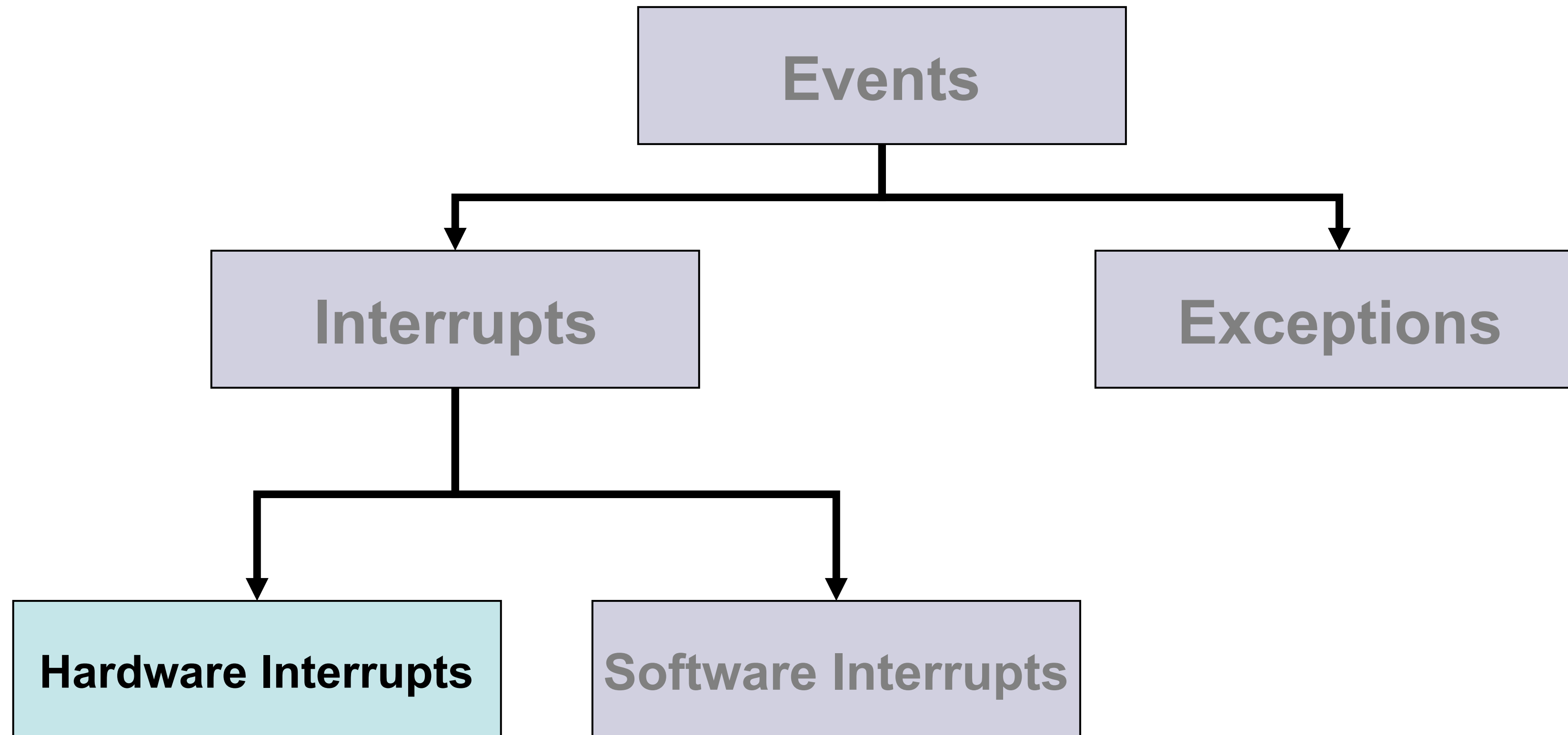
CR

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



CR

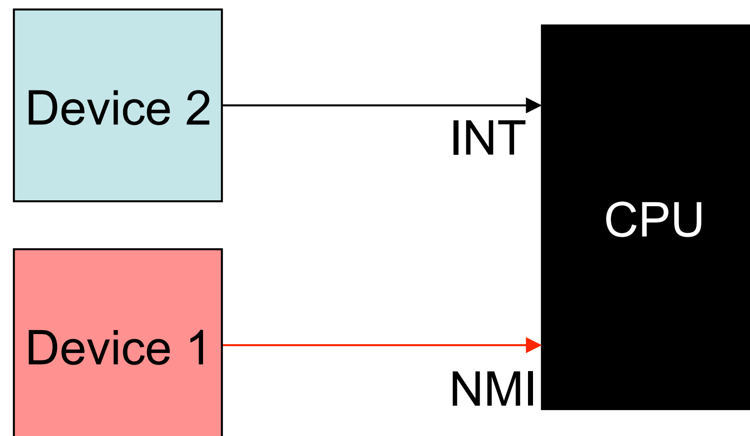
Why Hardware Interrupts?

- 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

CR

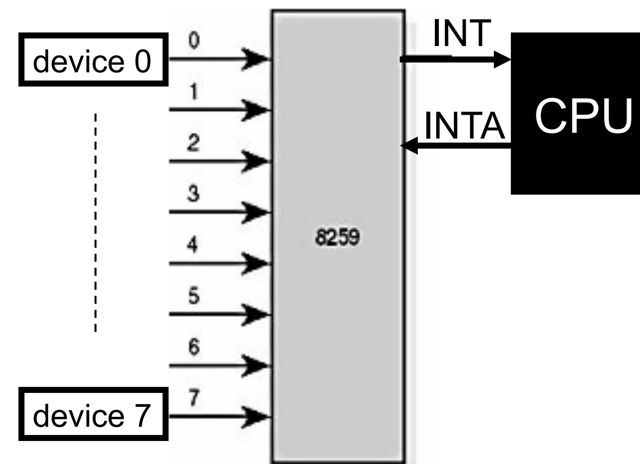
Interrupts

- 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?



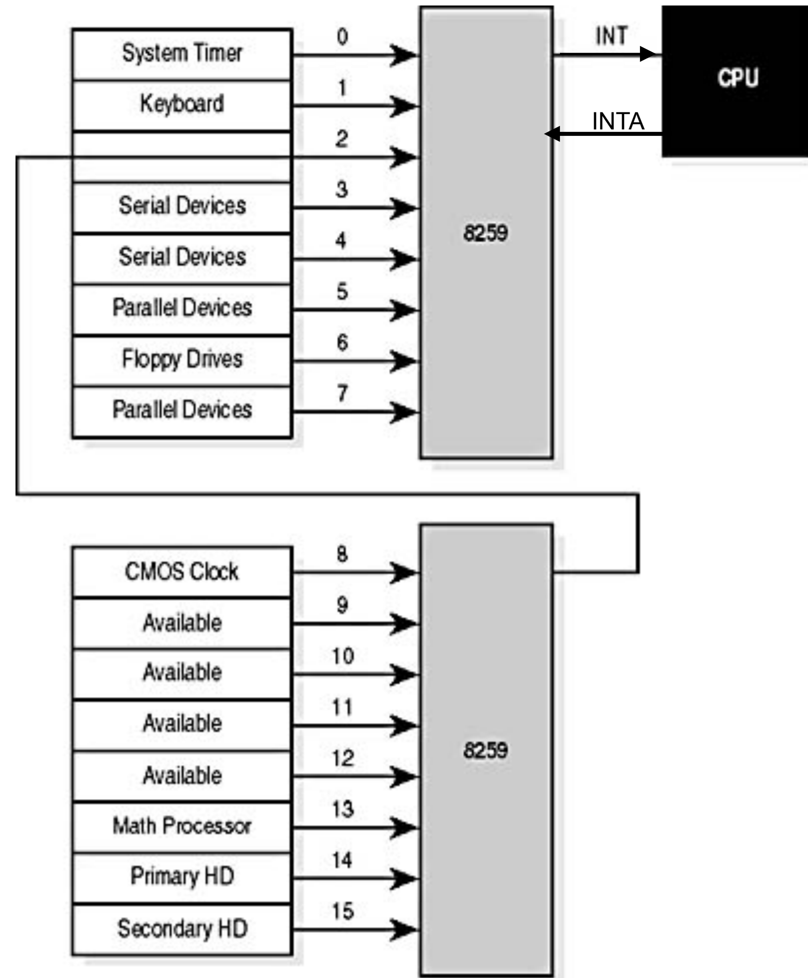
8259 Programmable Interrupt Controller

- 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



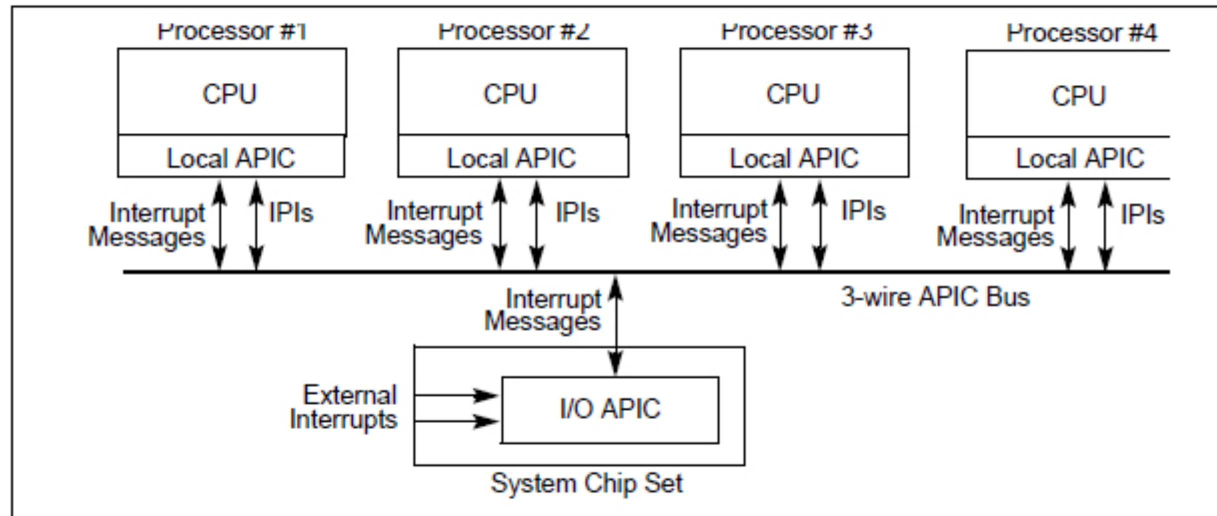
Interrupts in legacy CPUs

- 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
  - Multi-processor support is limited



CR

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

CR

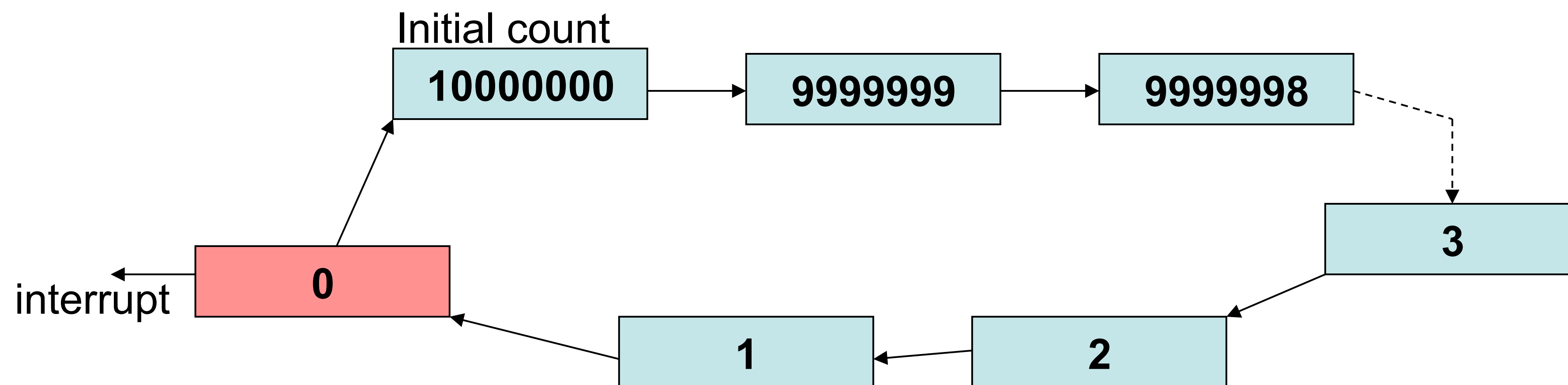
I/O APIC Configuration in xv6

- 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)

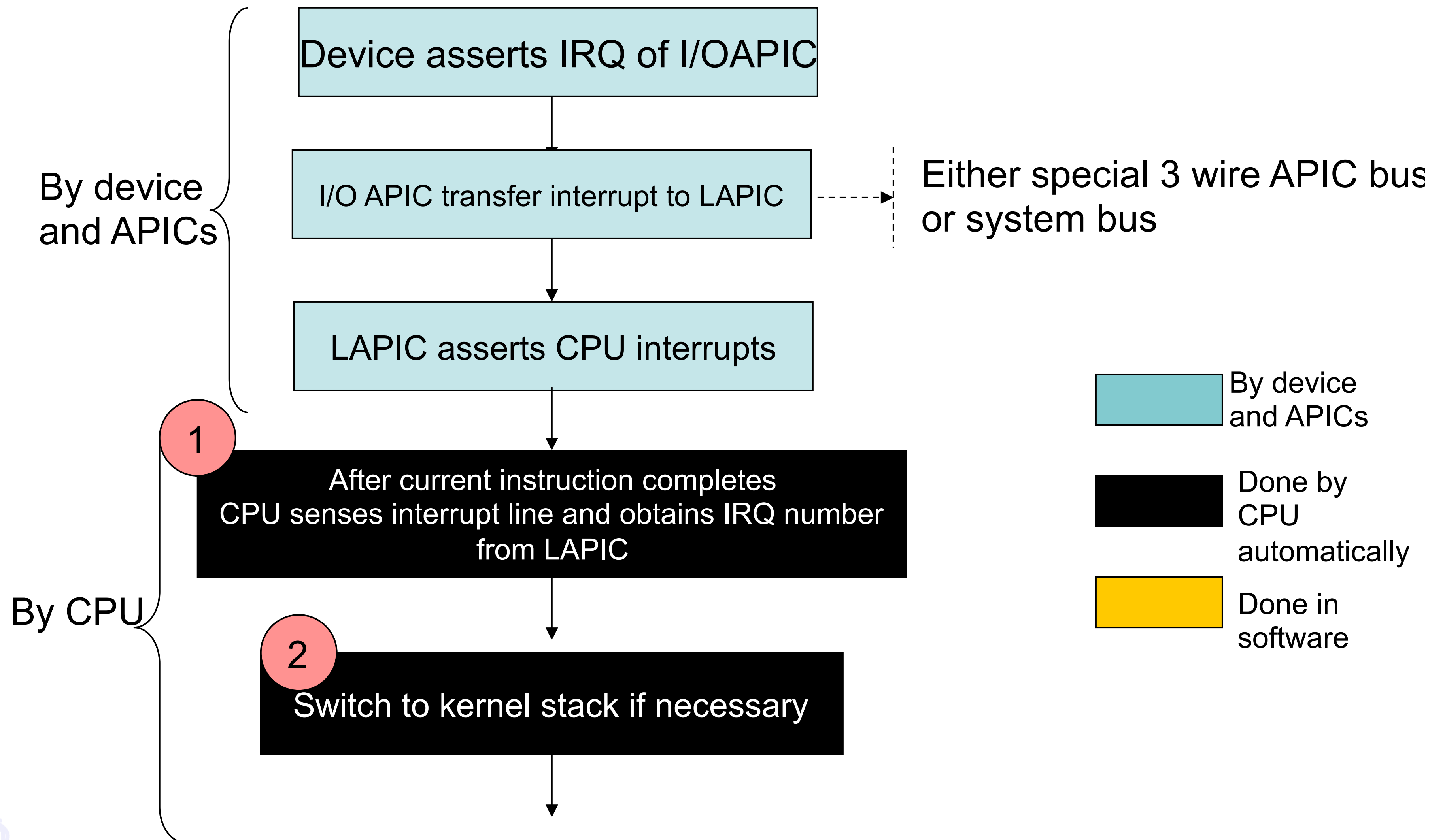
LAPIC Configuration in xv6

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



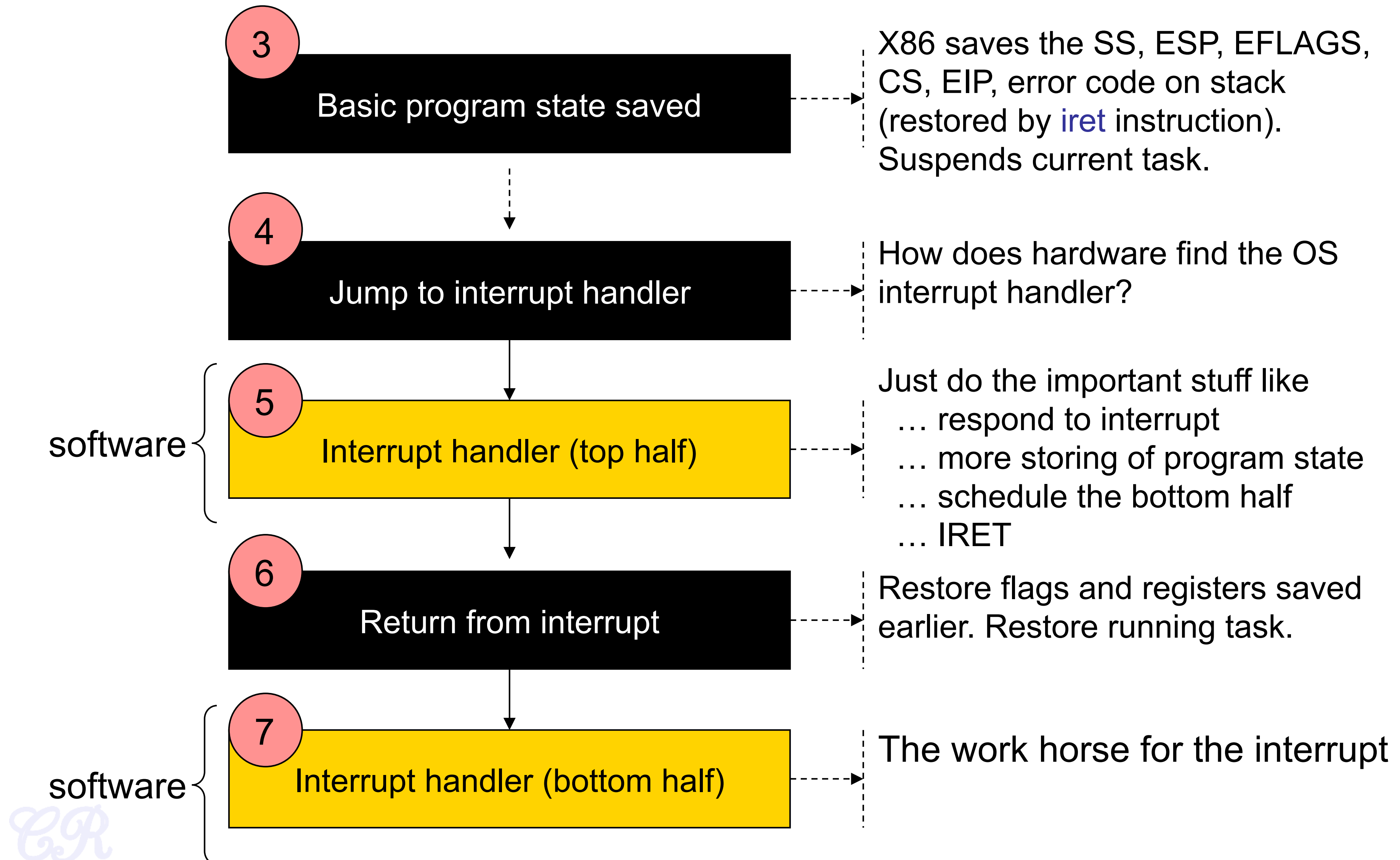
ref : lapic.c (lapicinit) (7151)

What happens when there is an Interrupt?



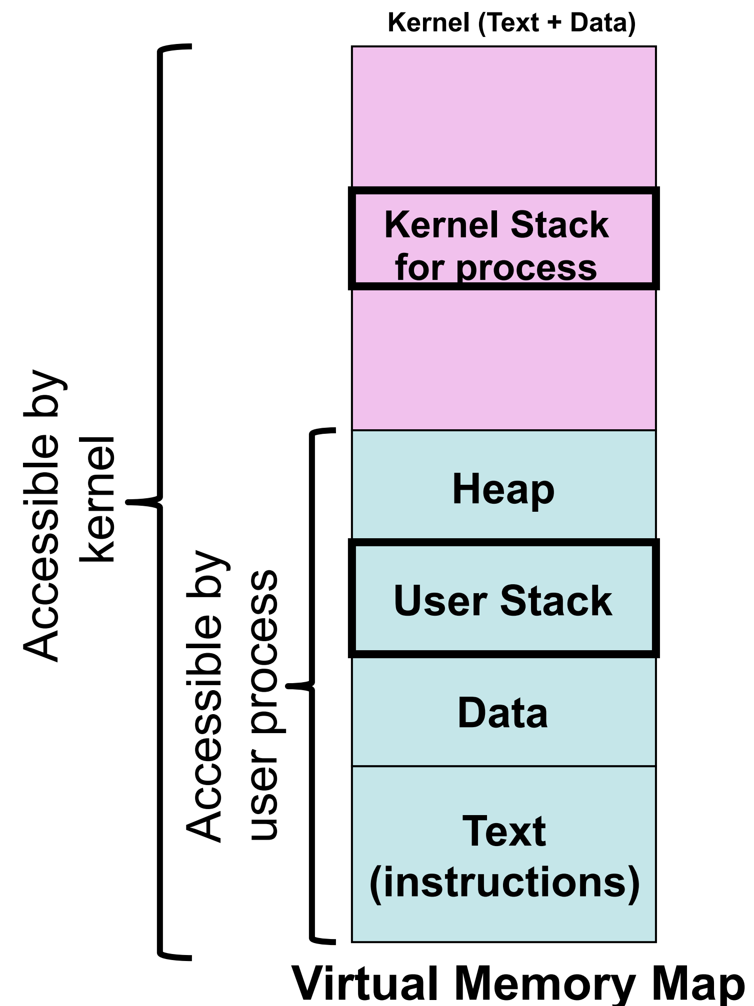
CR

What more happens when there is an Interrupt?



Stacks

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



CR

Switching Stack (to switch or not to switch)

2

- 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

CR

To Switch or not to Switch

Executing in Kernel space

- No stack switch
- Use the current stack

Executing in User space

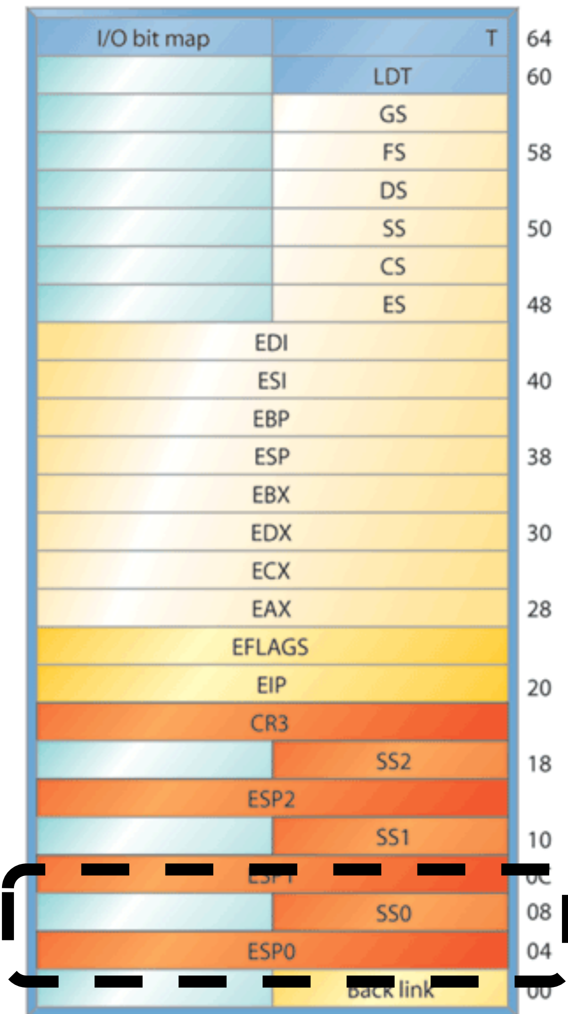
- Switch stack to a kernel switch

CR

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?

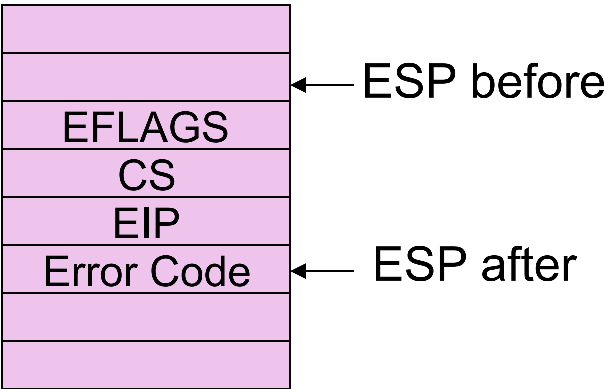
- Current program being executed must be able to resume after interrupt service is completed

CR

③ Saving Program State

Done automatically by CPU

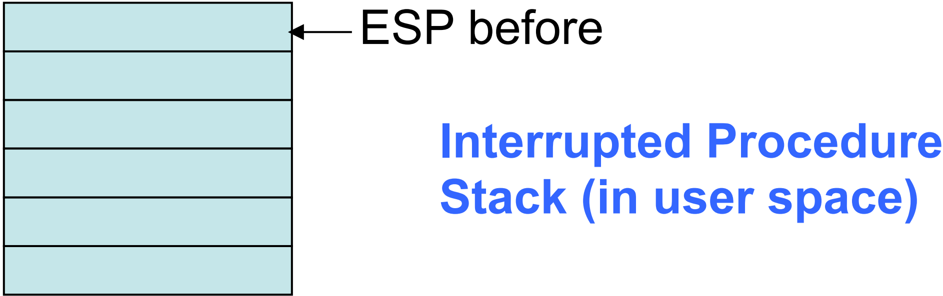
When no stack switch occurs
use existing stack



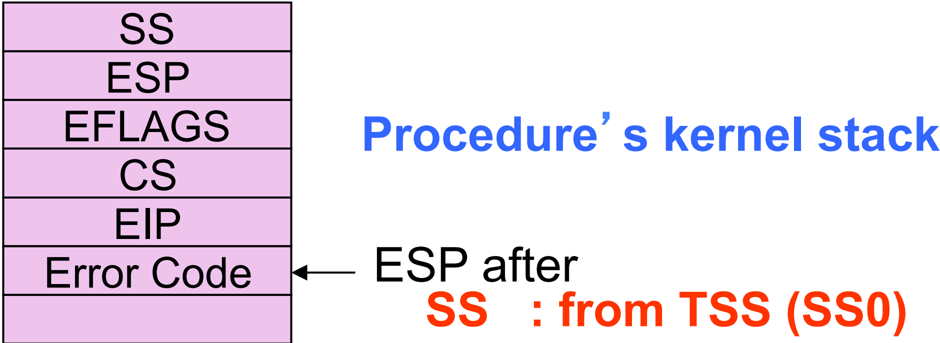
SS : No change
ESP : new frame pushed

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

When stack switch occurs
also save the previous SS and ESP



Interrupted Procedure Stack (in user space)



Procedure's kernel stack

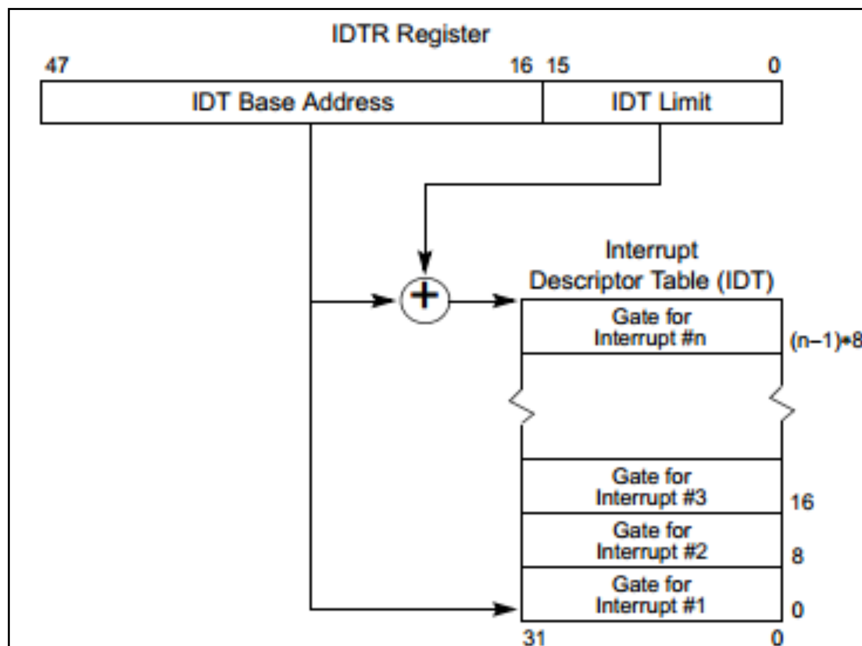
SS : from TSS (SS0)
ESP : from TSS (ESP0)

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)

CR

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

31–16	15	14–13	12–8	7–5	4–0	
Offset 31..16	P	DPL	0 D 1 1 0	0 0 0		4

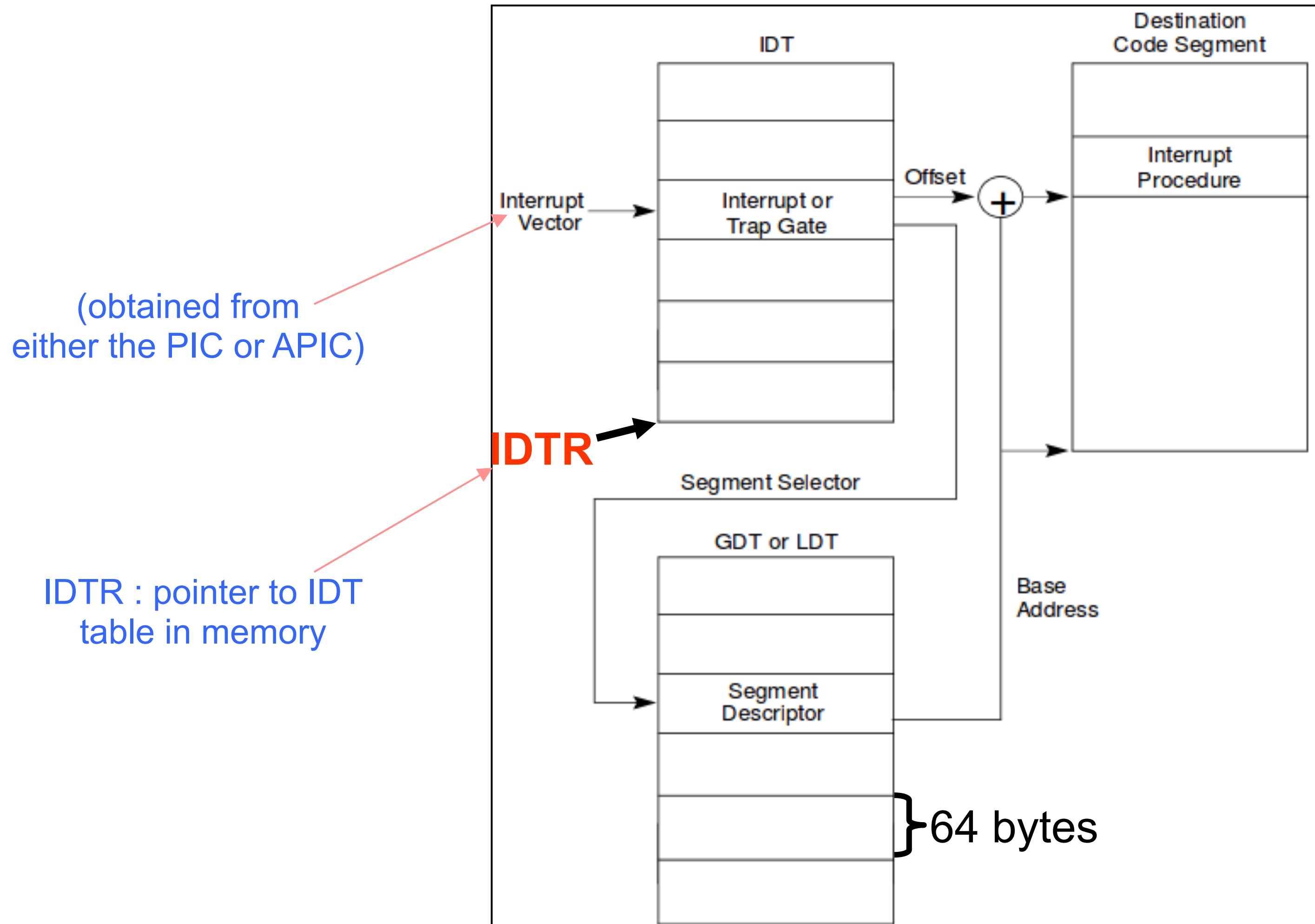
31–16	15–0	
Segment Selector	Offset 15..0	0

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)

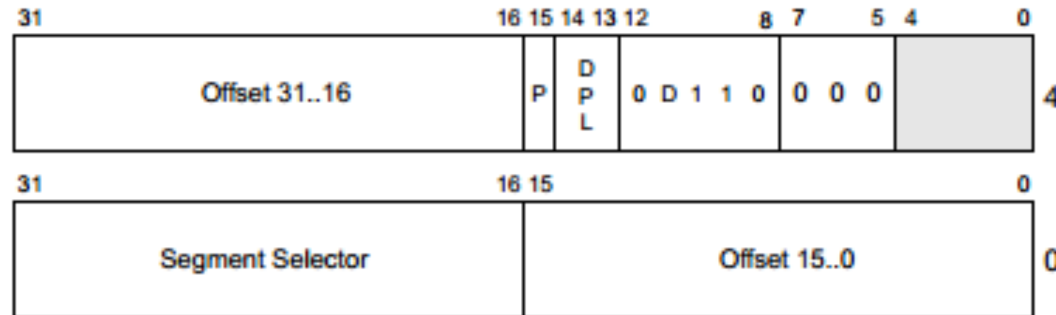
ref : SETGATE (0921), gatedesc (0901)

Getting to the Interrupt Procedure



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() (3329) in trap.c

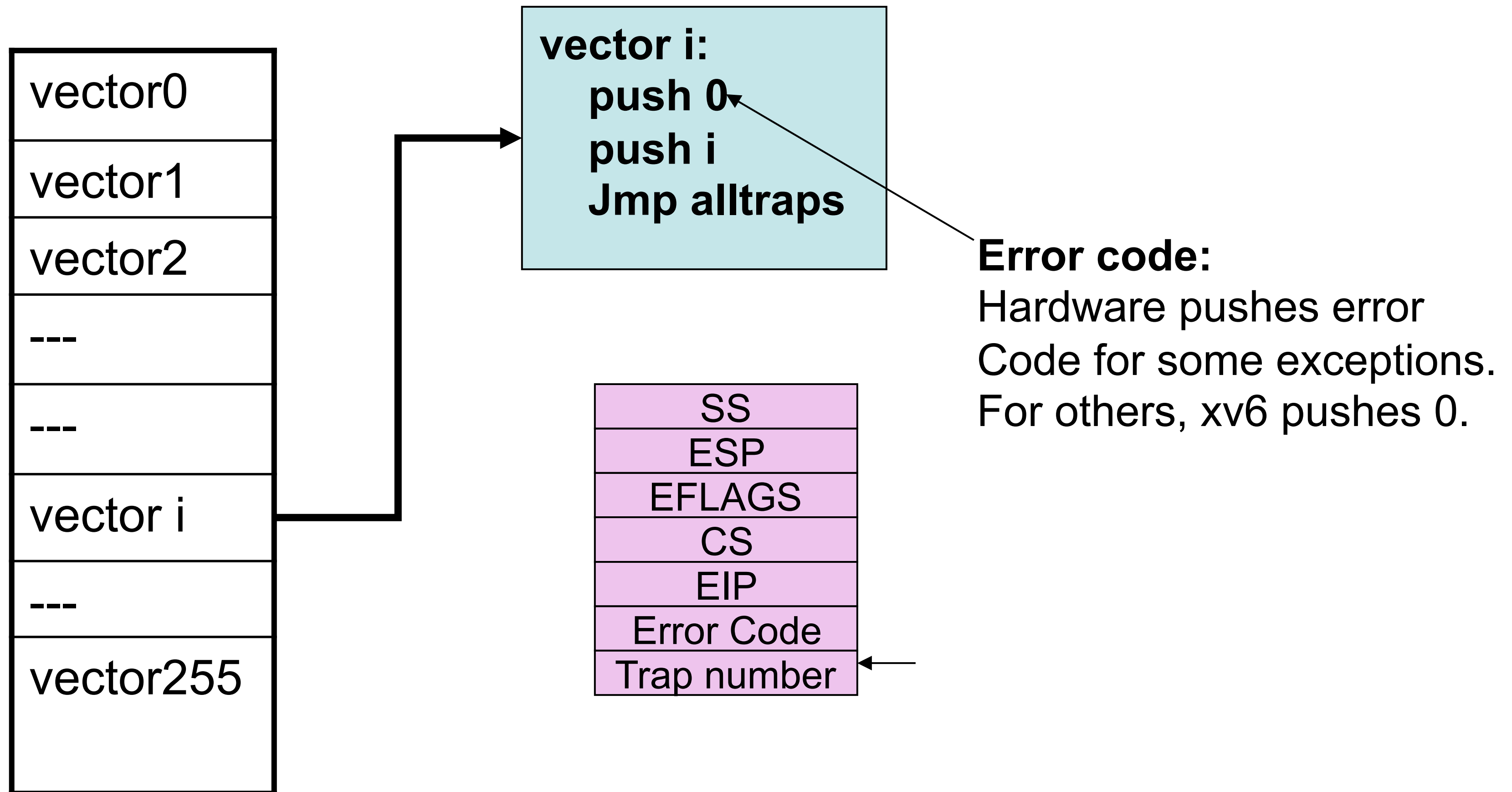
Setting up IDT in xv6

tvinit invoked from main; idtinit invoked from mpmain [12]

```
3310 // Interrupt descriptor table (shared by all CPUs).
3311 struct gatedesc idt[256];
3312 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3313 struct spinlock tickslock;
3314 uint ticks;
3315
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
3327
3328 void
3329 idtinit(void)
3330 {
3331   lidt(idt, sizeof(idt));
3332 }
```

```
0913 // Set up a normal interrupt/trap gate descriptor.
0914 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0915 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0916 // - sel: Code segment selector for interrupt/trap handler
0917 // - off: Offset in code segment for interrupt/trap handler
0918 // - dpl: Descriptor Privilege Level -
0919 //        the privilege level required for software to invoke
0920 //        this interrupt/trap gate explicitly using an int instruction.
0921 #define SETGATE(gate, istrap, sel, off, d)                \
```

Interrupt Vectors in xv6



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

5

alltraps

```
3253 .globl alltraps
3254 alltraps:
3255   # Build trap frame.
3256   pushl %ds
3257   pushl %es
3258   pushl %fs
3259   pushl %gs
3260   pushal
3261
3262   # Set up data and per-cpu segments.
3263   movw $(SEG_KDATA<<3), %ax
3264   movw %ax, %ds
3265   movw %ax, %es
3266   movw $(SEG_KCPU<<3), %ax
3267   movw %ax, %fs
3268   movw %ax, %gs
3269
3270   # Call trap(tf), where tf=%esp
3271   pushl %esp
3272   call trap
3273   addl $4, %esp
3274
3275   # Return falls through to trapret...
3276 .globl trapret
3277 trapret:
3278   popal
3279   popl %gs
3280   popl %fs
3281   popl %es
3282   popl %ds
3283   addl $0x8, %esp  # trapno and errcode
3284   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

SS

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

only if stack changed

By hardware

By software

Pushed by hardware or software

argument for trap
(pointer to this trapframe)

ESP

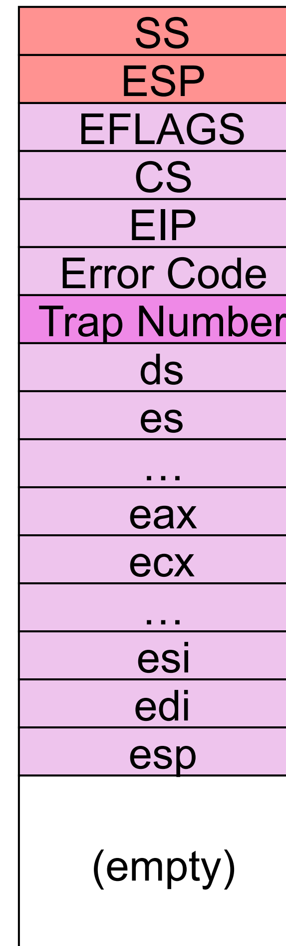
p->kstack

trapframe

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

trapframe struct

```
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // useless & ignored
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
0612
0613   // rest of trap frame
0614   ushort gs;
0615   ushort padding1;
0616   ushort fs;
0617   ushort padding2;
0618   ushort es;
0619   ushort padding3;
0620   ushort ds;
0621   ushort padding4;
0622   uint trapno;
0623
0624   // below here defined by x86 hardware
0625   uint err;
0626   uint eip;
0627   ushort cs;
0628   ushort padding5;
0629   uint eflags;
0630
0631   // below here only when crossing rings, such as from user to kernel
0632   uint esp;
0633   ushort ss;
0634   ushort padding6;
0635 };
```

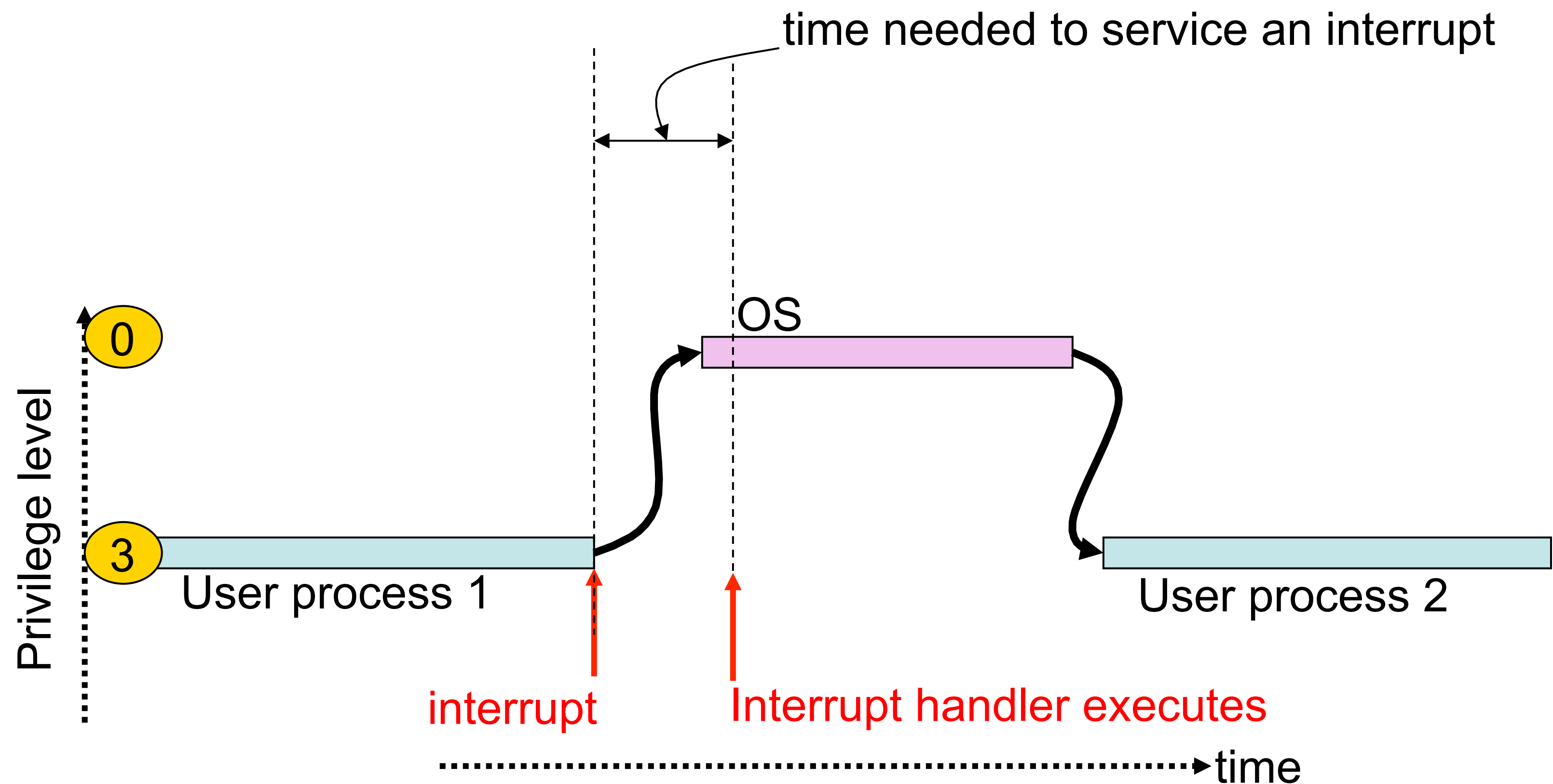


4 Interrupt Handlers

- 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)

CR

Interrupt Latency



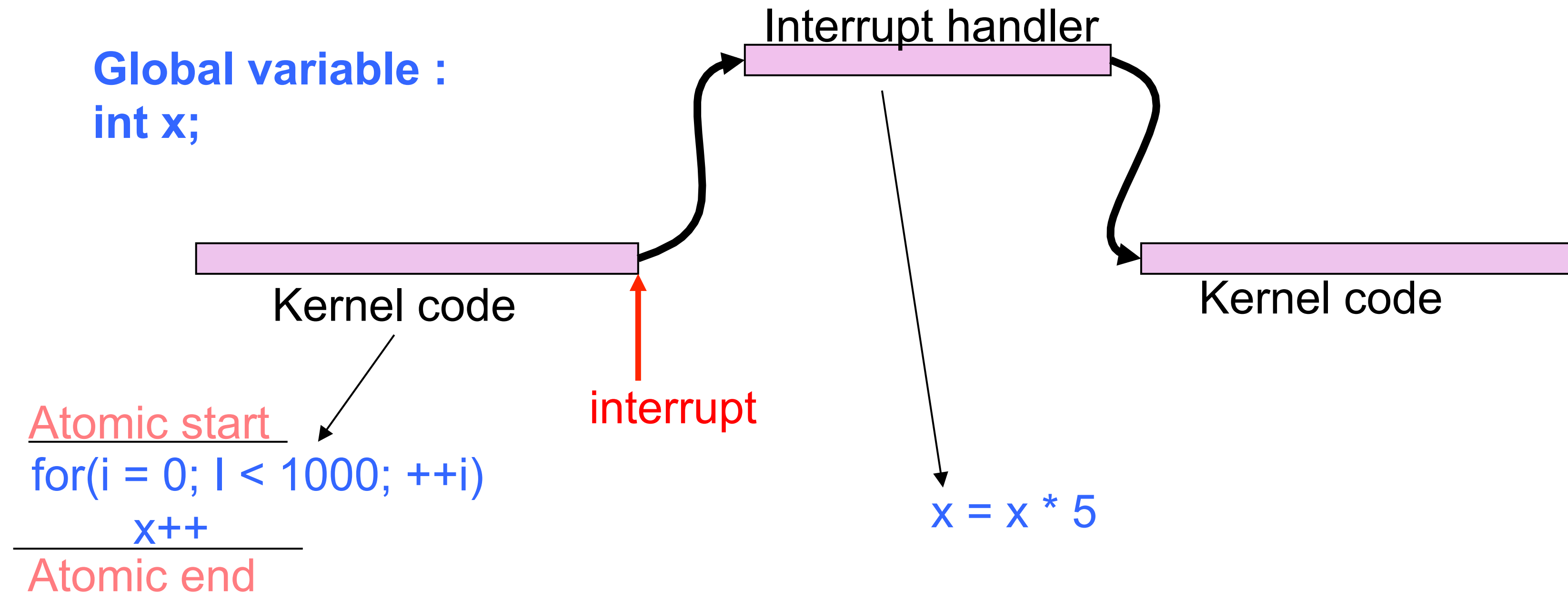
Interrupt latency can be significant

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.

CR

Atomic Operations

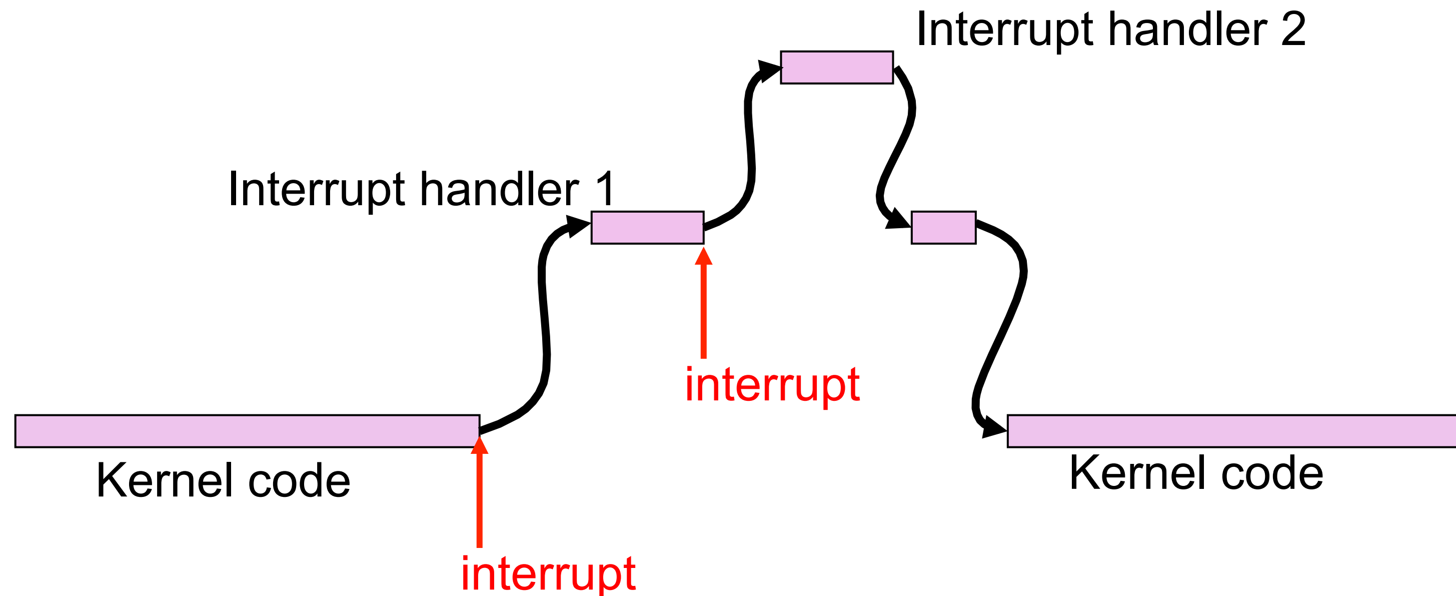


Value of x depends on whether an interrupt occurred or not!

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

CR

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
- **Linux Interrupt handler approach:** design interrupt handlers to be small so that nested interrupts are less likely

CR

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

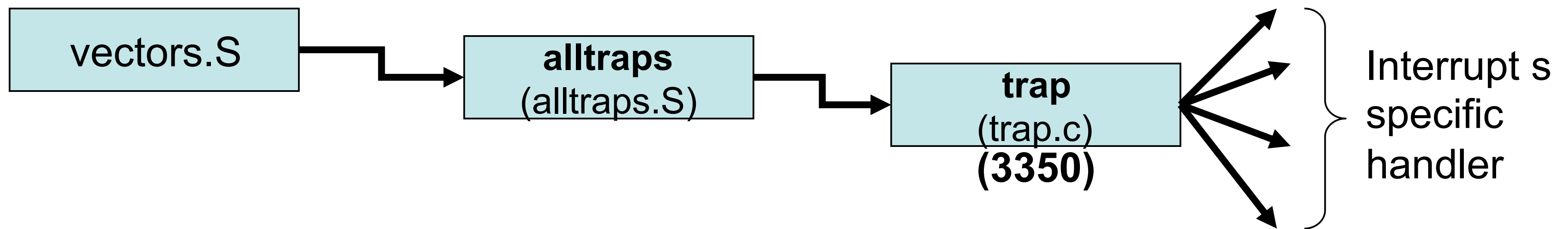
CR

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

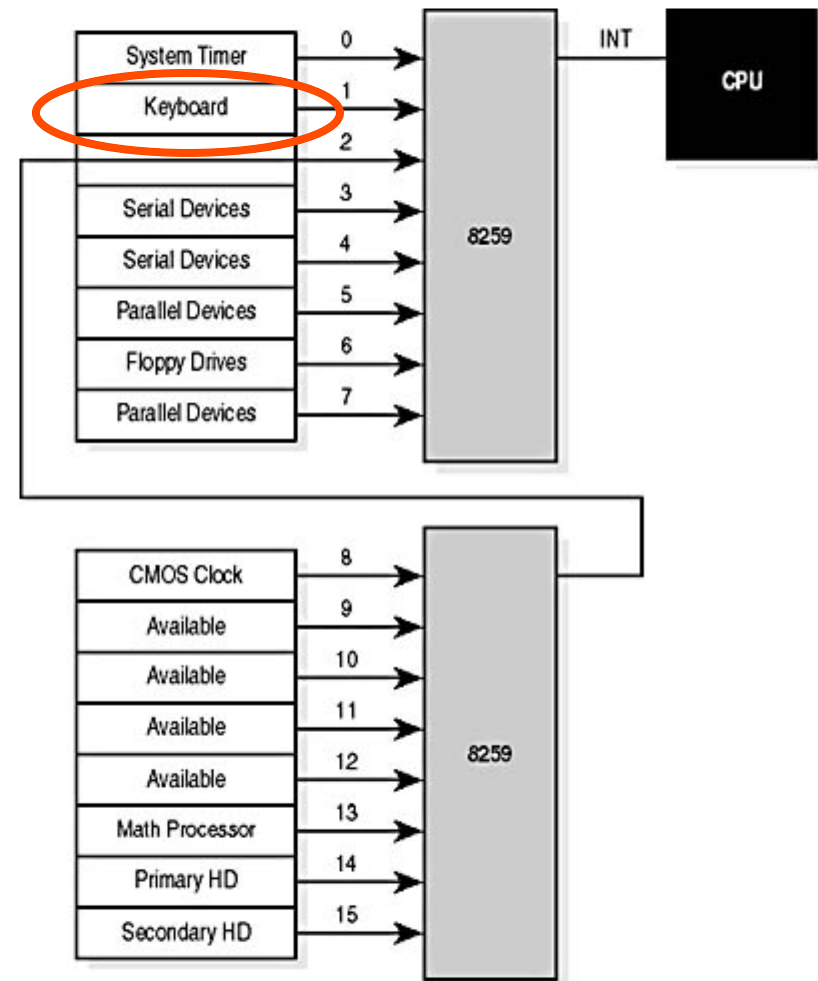
CR

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



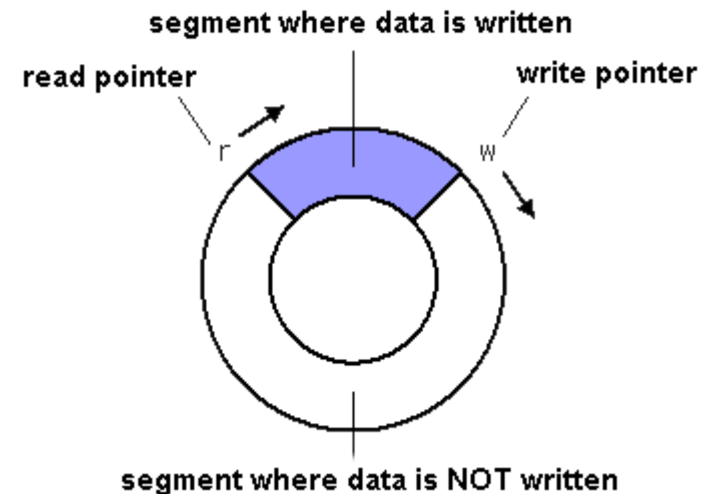
Keyboard Interrupt Handler

consoleintr (console.c)

get pressed character (kbdgetc (kbd.c0)) — talks to keyboard through specific predifined io ports

Service special characters

Push into circular buffer

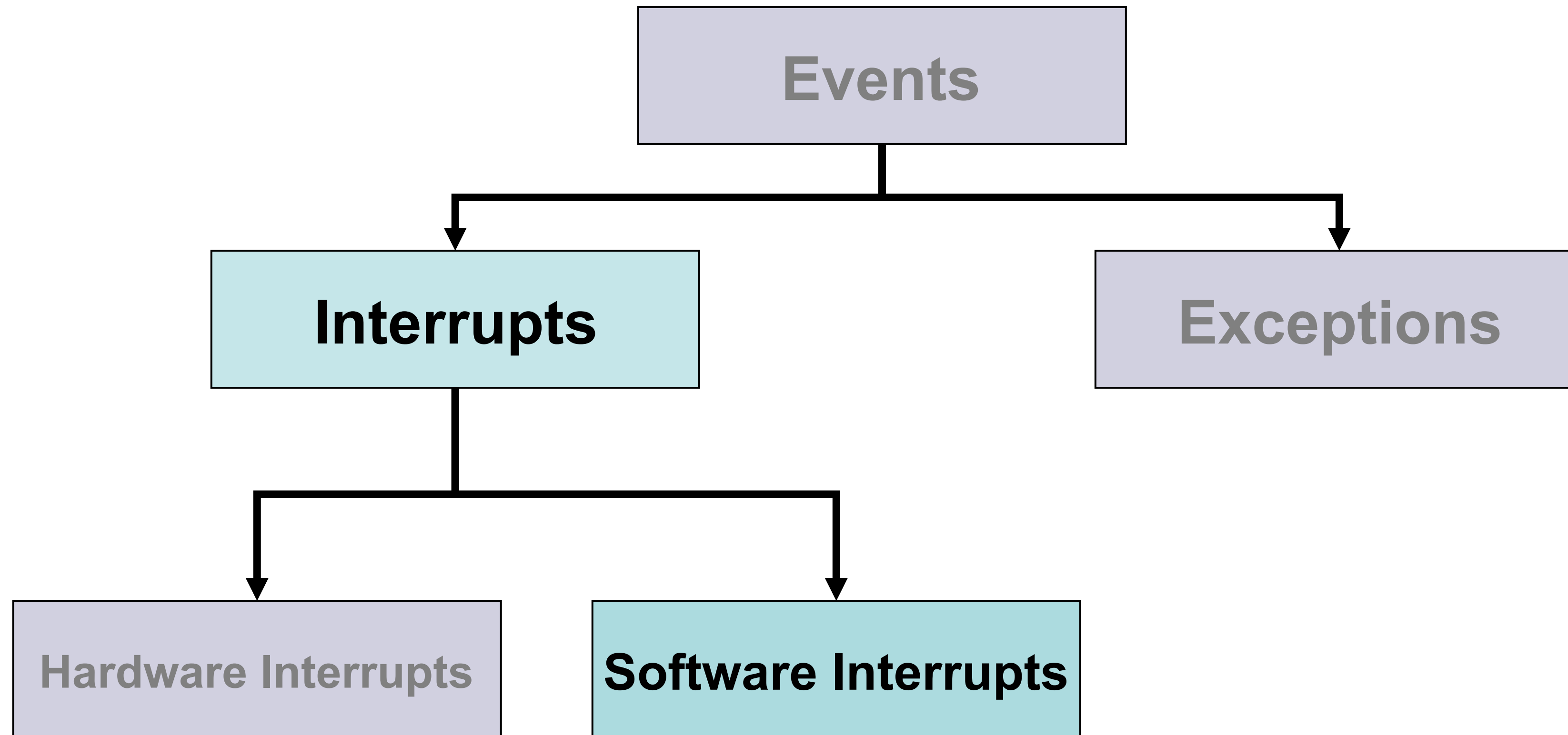


CR

System Calls and Exceptions

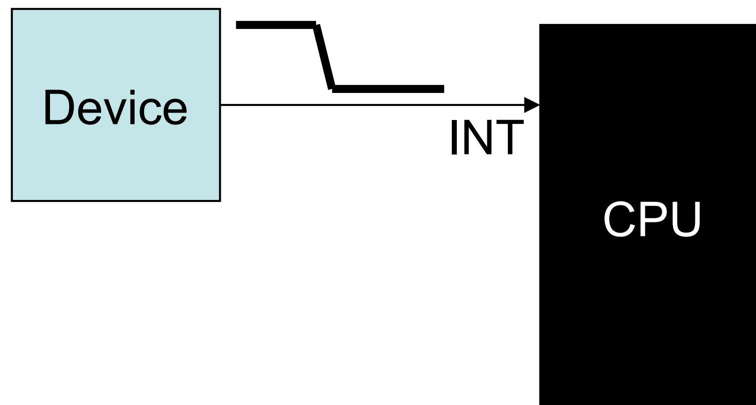
CR

Events



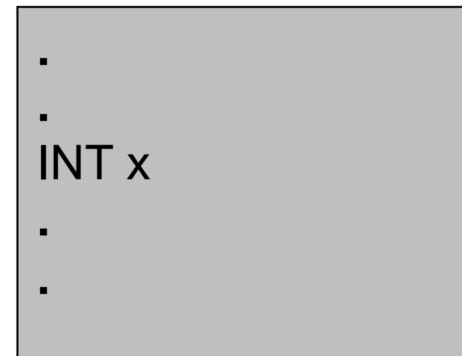
Hardware vs Software Interrupt

Hardware Interrupt



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

Software Interrupt



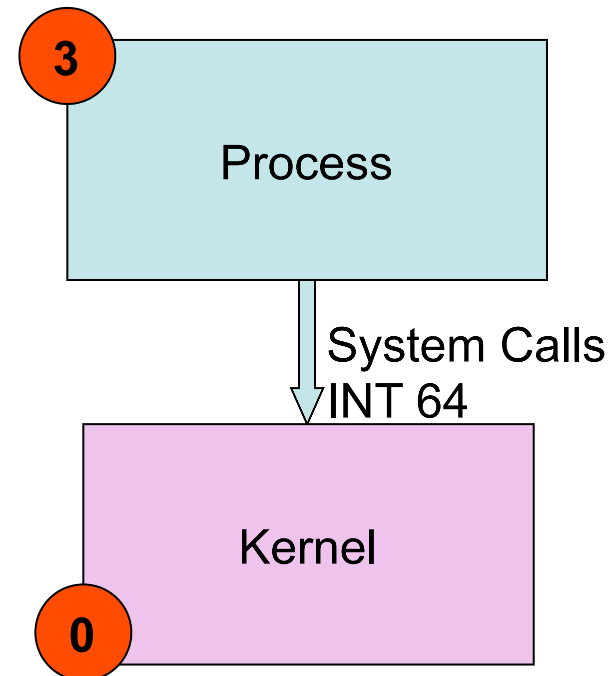
- An instruction which when executed causes an interrupt

CR

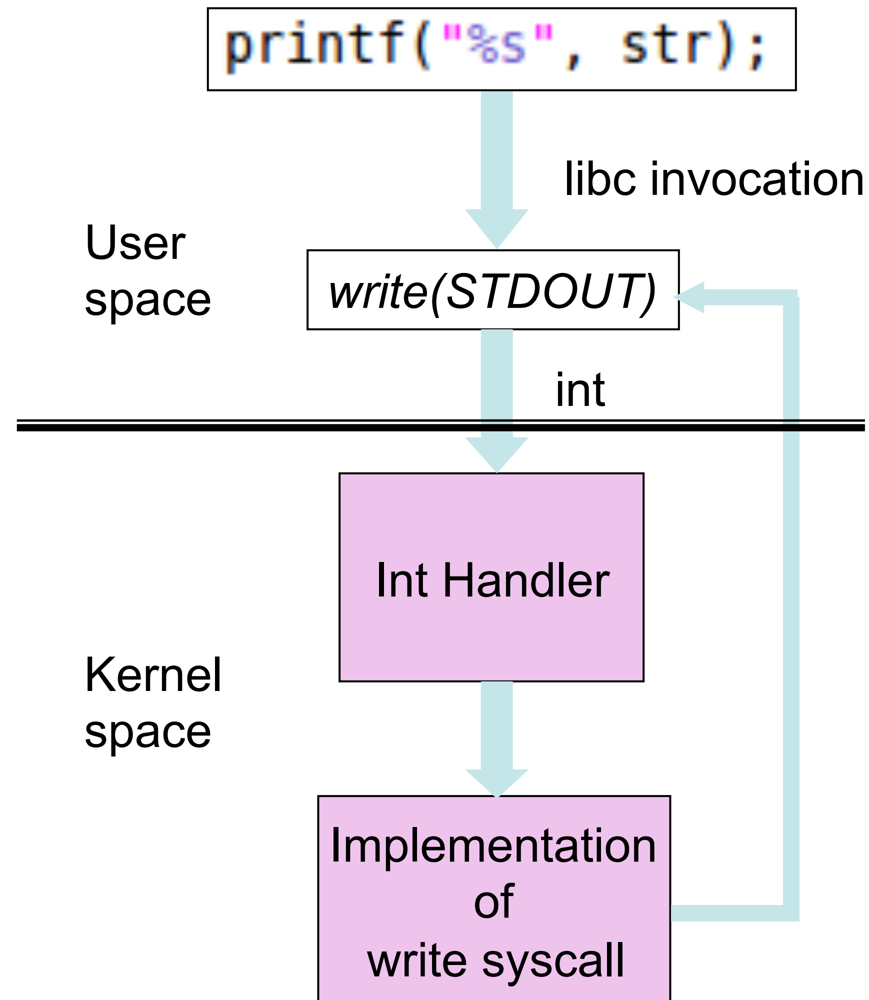
Software Interrupt

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

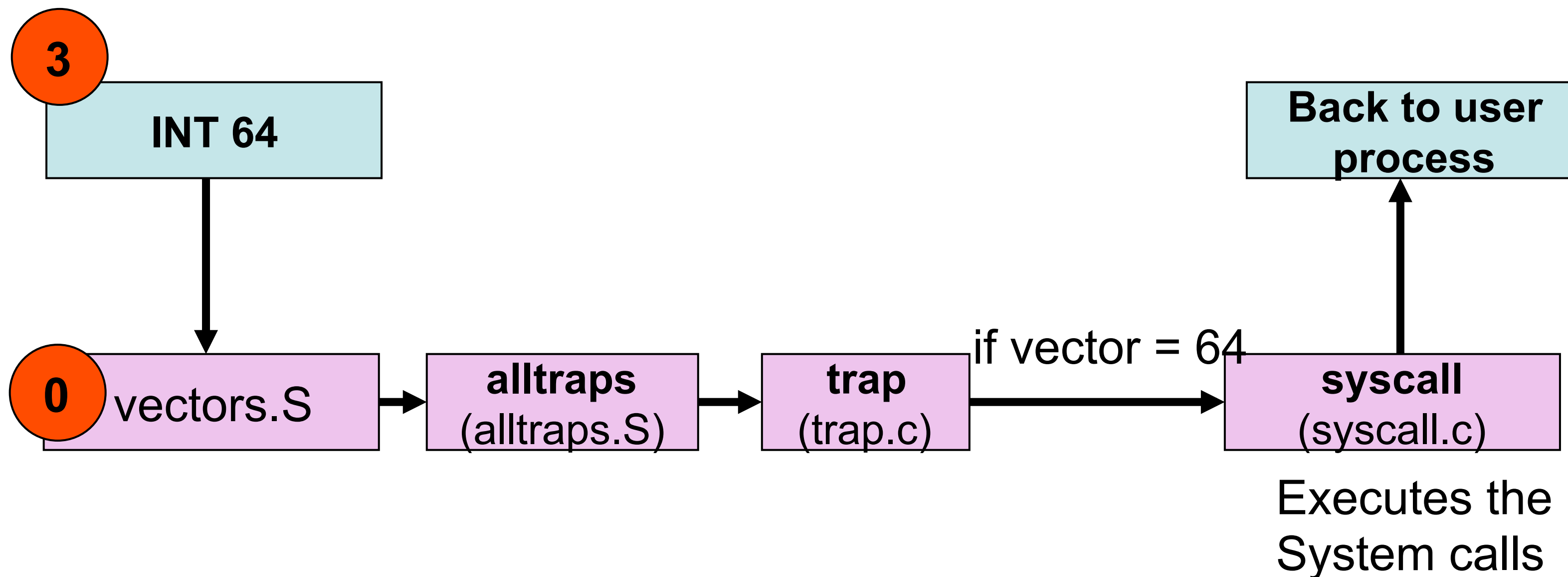


Example (write system call)



System call processing in kernel

Almost similar to hardware interrupts



CR

System Calls in xv6

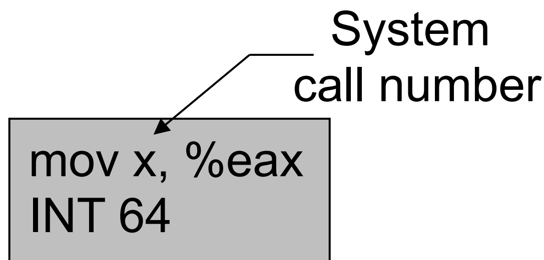
System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n 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?

CR

System Call Number

System call number used to distinguish between system calls



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,
```

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

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?

CR

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

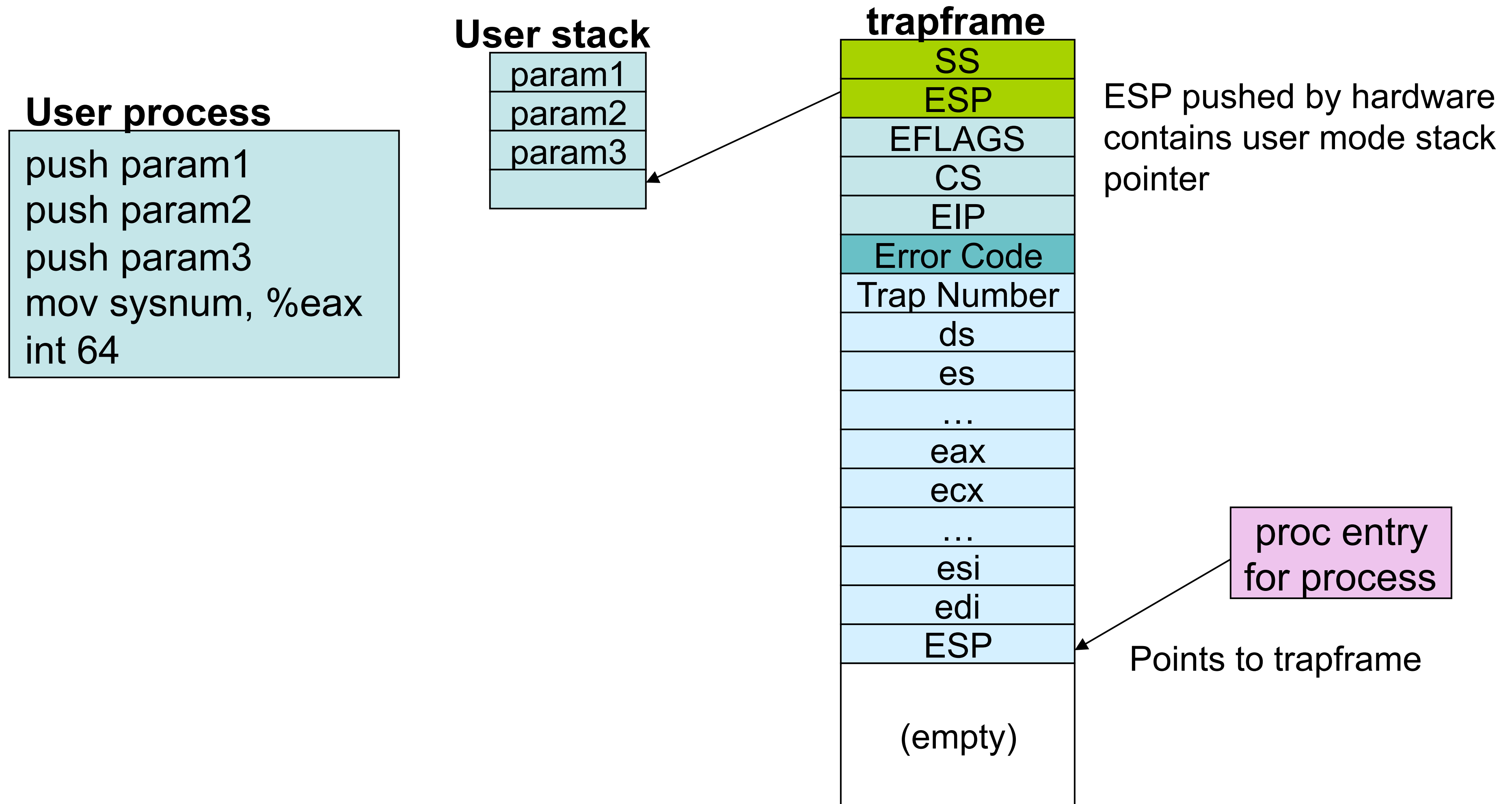
CR

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

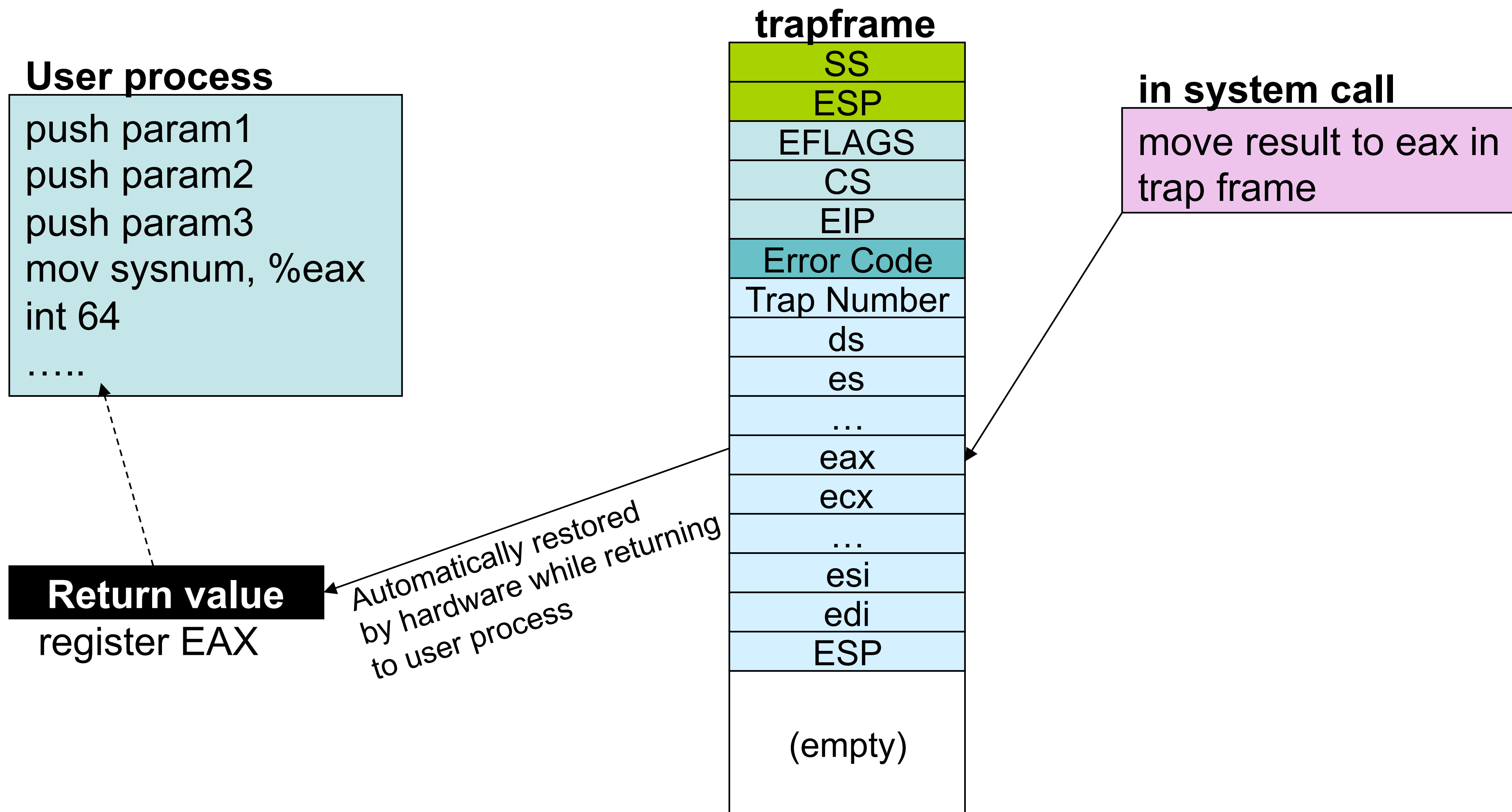
CR

Pass via User Mode Stack (xv6)

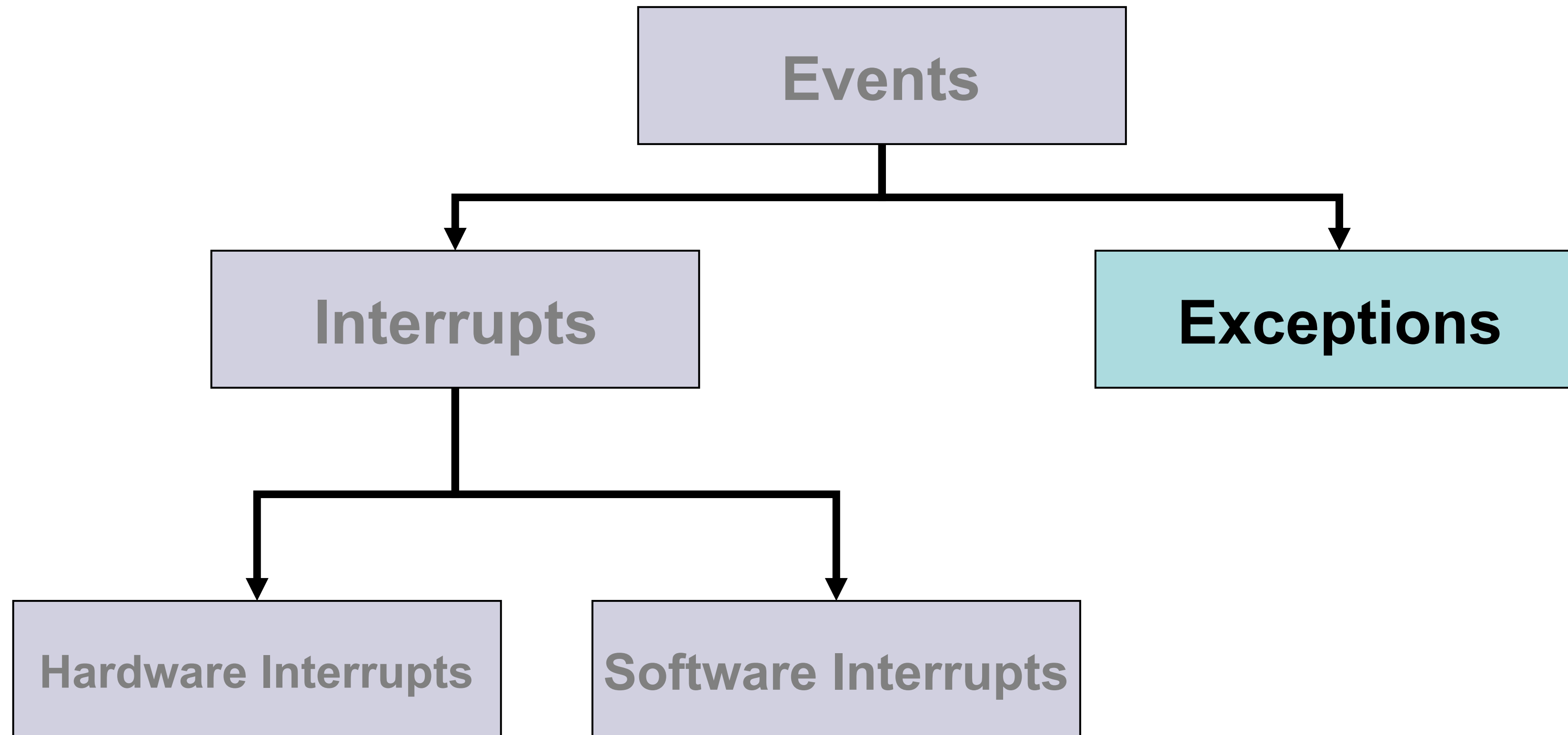


ref : sys_open (sysfile.c), argint, fetchint (syscall.c)

Returns from System Calls



Events



Exception Sources

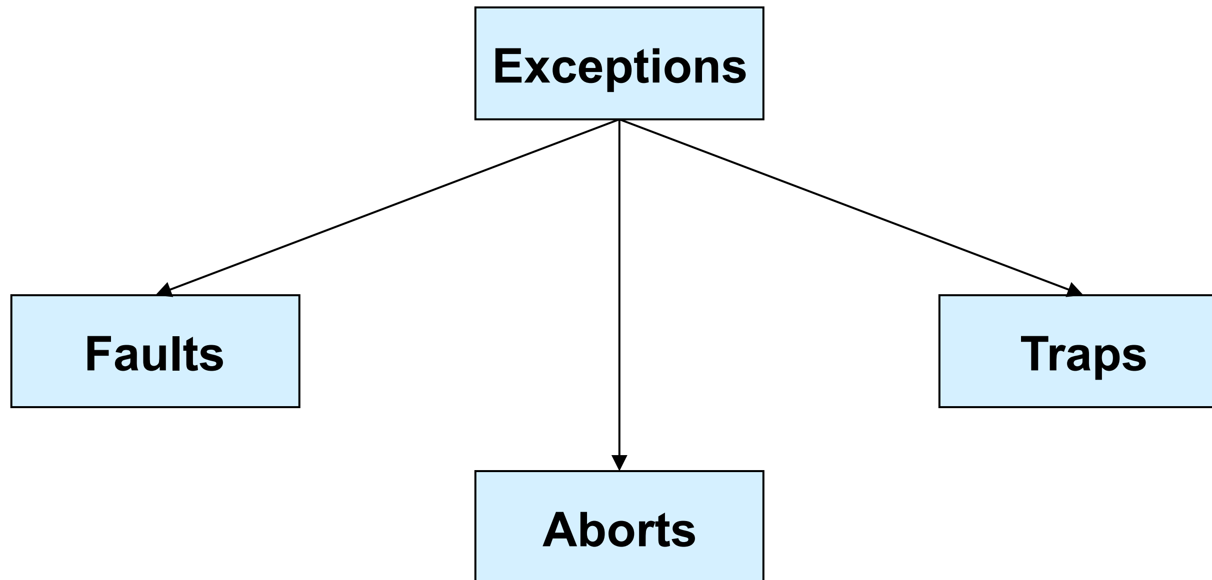
- Program-Error Exceptions
  - Eg. divide by zero
- Software Generated Exceptions
  - Example INTO, INT 3, BOUND
  - INT 3 is a break point exception
  - INTO overflow instruction
  - BOUND, Bound range exceeded
- Machine-Check Exceptions
  - Exception occurring due to a hardware error (eg. System bus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

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

Microsoft Windows : Machine check exception

CR

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

CR

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.

CR