Sistemi operativi

Lez. 23-24

Eccezioni, interrupt, syscall in JOS

© Danilo Bruschi

IL SUPPORTO HW

© Danilo Bruschi

Il ciclo ‘fetch-decode-execute

- Il processore opera costantemente sotto il controllo del seguente ciclo:

  1) Fetch the next instruction from ram
  2) Interpret the instruction just fetched
  3) Execute this instruction as decoded
  4) Advance the cpu instruction-pointer
  5) Go back to step 1

© Danilo Bruschi

Ma …

- Possono verificarsi circostanze che impongono al processore delle deroghe rispetto a questo comportamento, come ad esempio:
  - Quando durante l'esecuzione di un' istruzione si rilevano degli errori (Dati errati o non disponibili, accessi non autorizzati)
  - Un dispositivo esterno necessita di un intervento da parte del processore
  - Il processore deve svolgere un'attività diversa da quella programmata

Fault

- Se la CPU rileva, in fase di decode/execute che un'istruzione non può essere eseguita, il ciclo fetch-execute deve essere interrotto:
  - Questo tipo di errore è noto come 'fault'
- In questo caso il sistema reagisce:
  - Salvando alcune informazioni in opportune zone di memoria,
  - Cedendo il controllo ad una routine di fault-handling

© Danilo Bruschi

Fault-Handling

- In alcuni casi le causa che hanno portato al fault possono essere “rimediate” (ad es.: leggere/scrivere su un segmento “non presente”)
  - In questo caso, dopo gli opportuni interventi, il processore riprenderà il ciclo “fetch-execute” precedentemente interrotto
- In caso contrario il programma viene definitivamente interrotto, e in questo caso si parla più propriamente di ABORT invece di FAULT

Trap

- Un utente può anche programmare, durante l'esecuzione di un programma, il passaggio del controllo dal programma in esecuzione ad un altro programma, ad esempio
  - In fase di debugging di un programma A, è possibile programmare dei break point in modo che dopo l'esecuzione di un'istruzione di A, il controllo passi al debugger
- Questo tipo di situazione è denotata come TRAP
- Viene solitamente attivata dopo l'incremento di IP
- Come nel caso dei fault, quando si incontra un Trap il sistema provvede a:
  - salvare le informazioni necessarie per riprendere l'esecuzione del programma interrotto
  - Cedere il controllo al trap handler

Fault vs Trap

- Fault e Trap condividono una caratteristica comune, nell'esecuzione ripetuta di un programma, sotto le stesse condizioni iniziali, si verificano sempre nello stesso punto e sono quindi "predicibili"
- In questo senso si dice anche che si tratta si INTERRUZIONI SINCRONE
- Il sistema risponde in un modo molto simile al verificarsi di trap e fault, anche se le informazioni che devono essere salvate sono diverse

© Danilo Bruschi

Fault vs Trap

- **FAULT**
  - L'indirizzo salvato deve essere quello dell'istruzione che ha provocato il fault, istruzione che dovrà essere ricaricata dopo che il problema che ha causato il fault è stato rimosso

- **TRAP:**
  - l'indirizzo salvato è quello dell'istruzione successiva a quella che ha provocato il trap

© Danilo Bruschi

Eccezioni

- Nell'architettura IA-32 fault e trap sono più genericamente accorpati nella categoria delle eccezioni, che possono essere di tre tipi :
  - Fault: una eccezione dovuta ad un errore che può essere corretto e può consentire, dopo la correzione, la ripresa del programma che l'ha generato. In questo caso sullo stack va salvato l'indirizzo della faulting instruction
  - Trap: un richiesta esplicita di intervento attraverso un'istruzione di trapping (INT)
  - Abort: una eccezione che non consente la ripresa del programma che l'ha provocata

© Danilo Bruschi

Eventi Asincroni

- Dispositivi “esterni” al processore possono avere la necessità di comunicare con lo stesso quando
  - sono state portate a termine operazioni precedentemente richieste dal processore per conto di processi
  - Il dispositivo ha assunto un nuovo stato (ready, faulty, ecc.) di cui è necessario avvertire il sistema
- I dispositivi esterni operano indipendentemente dalla CPU e gli eventi che generano non possono essere predicibili
- In questo caso parliamo di eventi asincroni

© Danilo Bruschi

La gestione degli eventi asincroni

- Problema: i dispositivi esterni operano “svincolati” dal processore e hanno la necessità di comunicare in istanti che non è possibile prestabilire
- Desiderata: una modalità che consenta al processore di gestire eventi che non può predire
  - Il processore deve poter eseguire altre attività mentre è in attesa di un evento asincrono
  - Gli eventi devono essere gestiti velocemente e a basso costo (low overhead)

© Danilo Bruschi

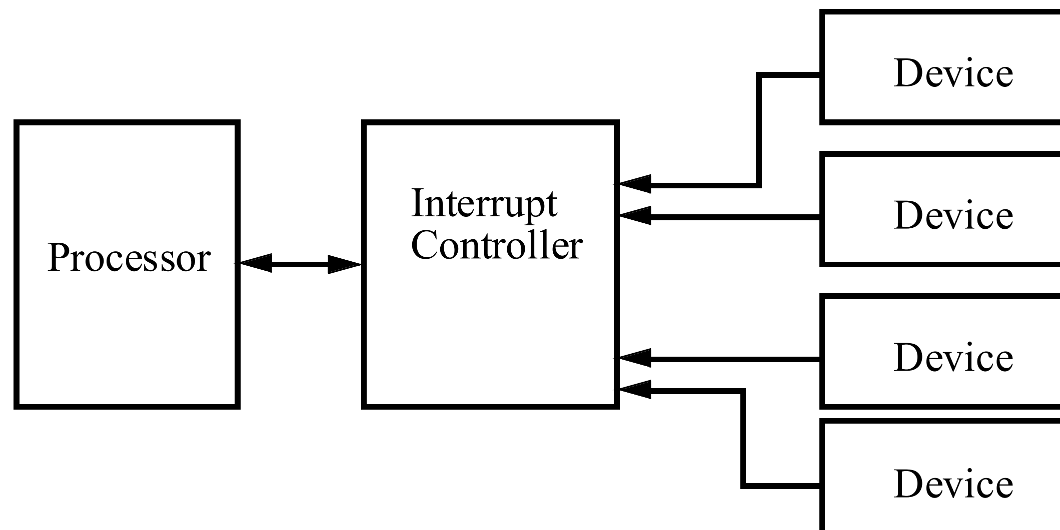
Alternativa 1: Polling

- Periodicamente il processore verifica se ci sono richieste pendenti da parte dei dispositivi esterni:
  - Spreca cicli di CPU anche quando non ci sono richieste pendenti
  - Il tempo medio di risposta può essere elevato per la gestione di eventi critici
  - Determinare la periodicità corretta è un problema molto critico

© Danilo Bruschi

Alternativa 2: Interrupt

- Fornire ogni dispositivo di un connessione (interrupt line) che può usare per comunicare fisicamente con il processore
  - Quando il processore "sente" un interrupt, il processore esegue una routine chiamata *interrupt handler*
  - No overhead quando non ci sono richieste pendenti



Interrupt Vector

- Ciascun interrupt ed eccezione è identificato da un numero compreso tra 0 e 255 chiamato, **vettore**
- I vettori 0-8, 10-14, e 16-19 sono interrupt ed eccezioni predefiniti, i vettori 32-255 sono a disposizione degli utenti e sono chiamati **maskable interrupts**
- Il flag IF del registro EFLAGS può disabilitare il servizio di interrupt mascherabili ricevuto sul pin INTR del processore
- Il flag IF viene gestito attraverso le istruzioni STI (set interrupt-enable flag) e CLI (clear interrupt-enable flag), eseguibili solo con opportuni privilegi

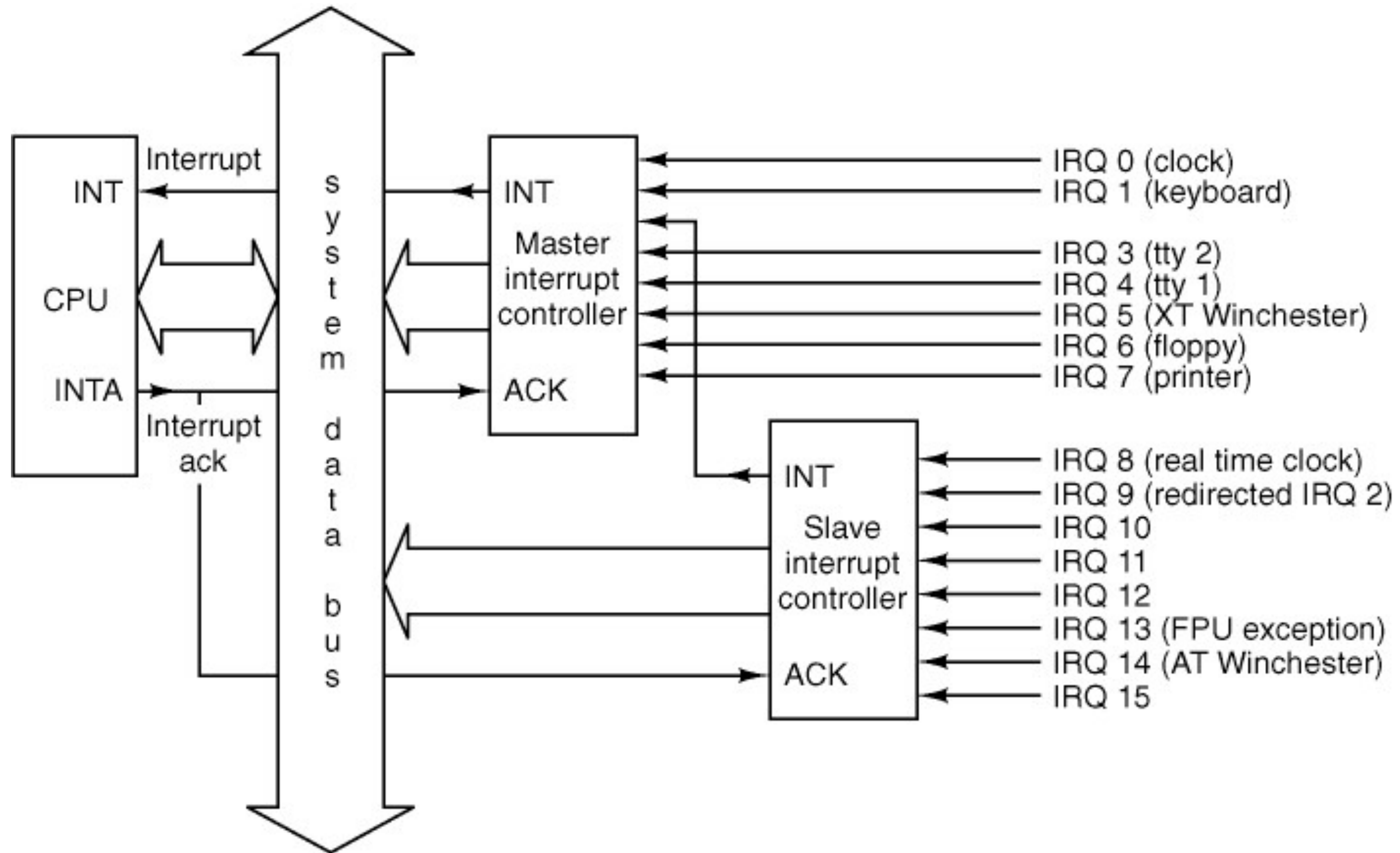
© Danilo Bruschi

Interrupt Controller

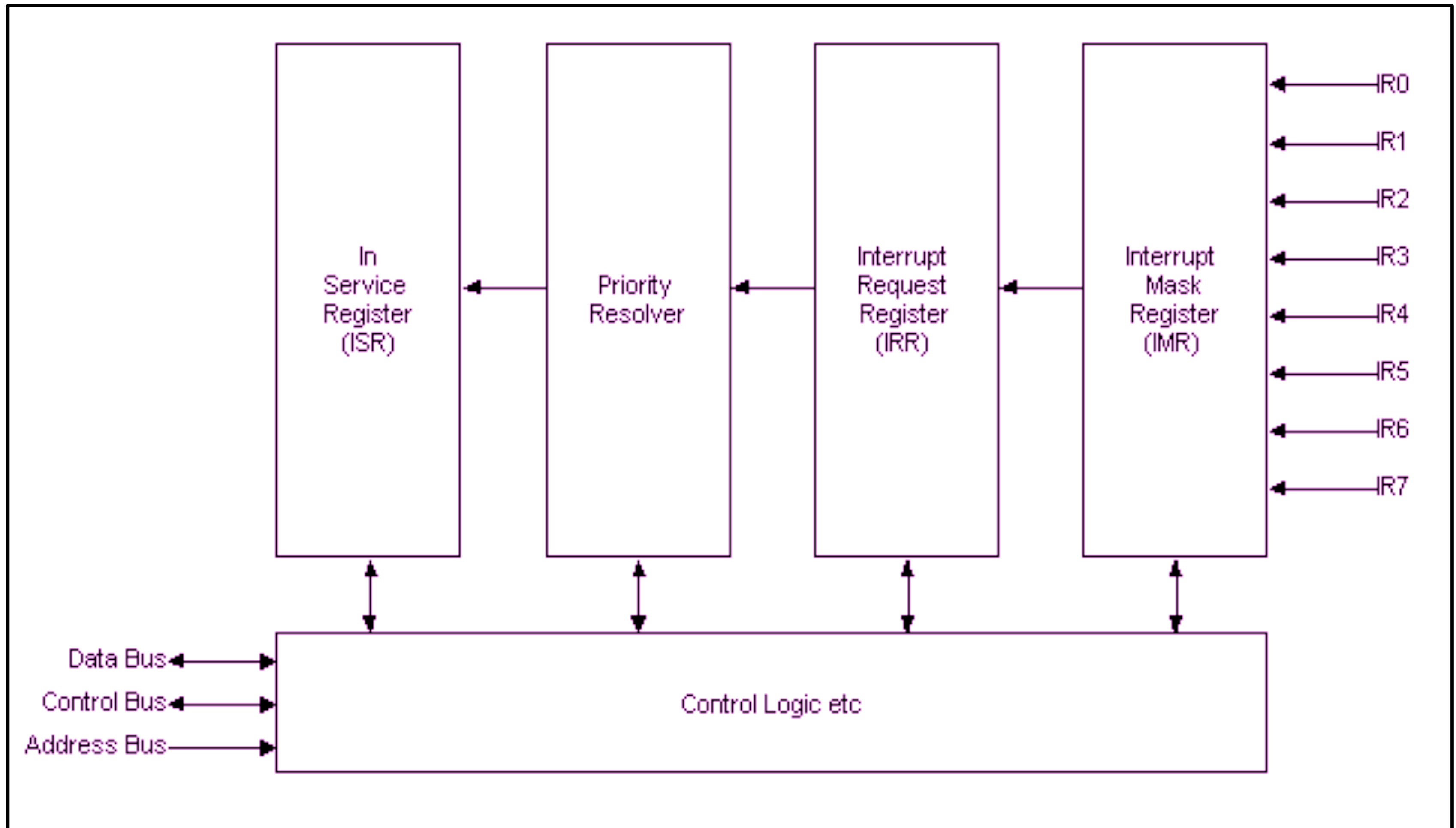
- Per la gestione dei diversi segnali di interrupt che provengono dai vari dispositivi si ricorre a hardware dedicato:
  - Stabilisce la priorità tra più interrupt pendenti
  - Segnala al processore quale interrupt servire per primo
- Questo hardware è l' Interrupt Controller

© Danilo Bruschi

Schema generale



Schema interrupt controller



© Danilo Bruschi

Funzionamento PIC

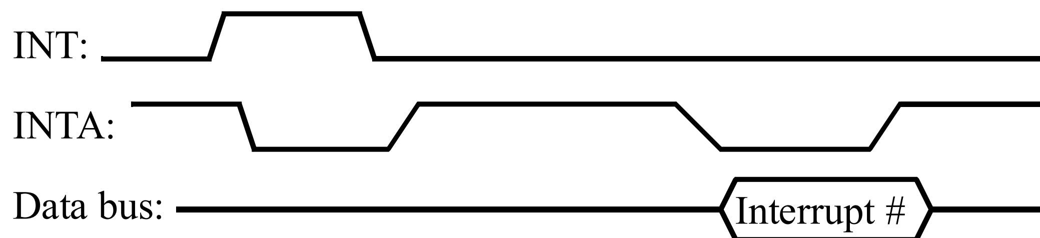
- Le linee di interrupt passano per prima cosa attraverso il registro IMR (Interrupt Mask Register) che verifica se le stesse sono state mascherate
- Nel caso di interrupt non mascherati, gli stessi sono registrati nel registro IRR (Interrupt Request Register), che li “conserva” finchè sono stati opportunamente gestiti
- Priority Resolver seleziona l’ IRQ di più alta priorità
- Viene inviato al processore il segnale INT

© Danilo Bruschi

Funzionamento PIC

80386 ha una sola linea di interrupt e una linea di interrupt acknowledge, il protocollo per la segnalazione di un interrupt è:

- Il processore, appena sentito l'interrupt, risponde con il segnale INTA (Interrupt Acknowledge)
- Alla ricezione del segnale INTA, il segnale IRQ che il PIC sta gestendo viene memorizzato nel registro ISR (In Service Register ISR). Il bit corrispondente all'IRQ viene anche resettato all'interno di IRR e vengono disabilitati tutti gli interrupt di priorità minore o uguale a quello in corso
- Il processore invia un altro segnale INTA, a seguito del quale il PIC carica sul data bus un numero di 8 bit che corrisponde al vettore dell'interrupt



Registri di comando del PIC

- Programmable Interrupt Controller (PIC) gestisce gli interrupt hardware. Esistono in genere due PIC in un PC, a due diversi indirizzi
- Il primo PIC, posizionato all'indirizzo 0x20h controlla gli IRQ da 0 a 7
- Il secondo PIC posizionato all'indirizzo 0xA0 controlla gli IRQ da 8 a 15

© Danilo Bruschi

Comandi del PIC

- Ai suddetti registri possono essere inviati comandi (usando l'istruzione out) che regolano il comportamento del PIC
- Ad esempio è possibile abilitare o disabilitare gli interrupt direttamente sul PIC invece che sul processore. La seguente sequenza abilita IRQ4:

```
in     0x21, %al        !Read existing bits.
and    %al, $0xef       !Turn on IRQ 4 (COM1).
out    %al, 0x21        !Write result back to PIC
```

- E se voglio disabilitare IRQ 2?

© Danilo Bruschi

EOI

- Un’altra istruzione estremamente importante nella gestione degli interrupt è l’istruzione EOI (End of Interrupt)
- Deve essere eseguita dal gestore dell’interrupt al termine della sua esecuzione e viene inviata dalla Cpu al PIC
- EOI consente la riabilitazione degli interrupt a livello PIC e consiste nell’invio del valore 0x20 al registro di comando interessato

```
mov     $0x20,%al
out     %al, 0x20
```


© Danilo Bruschi

Interrupt handling

- Il processore usa il vettore dell'interrupt per decidere quale handler attivare
- Il processore interrompe il processo corrente PROC, e ne salva lo stato
- Il processore salta a un interrupt handler
- Quando l'interrupt è stato gestito, lo stato di PROC viene ripristinato e PROC riprende l'esecuzione da dove era stato sospeso

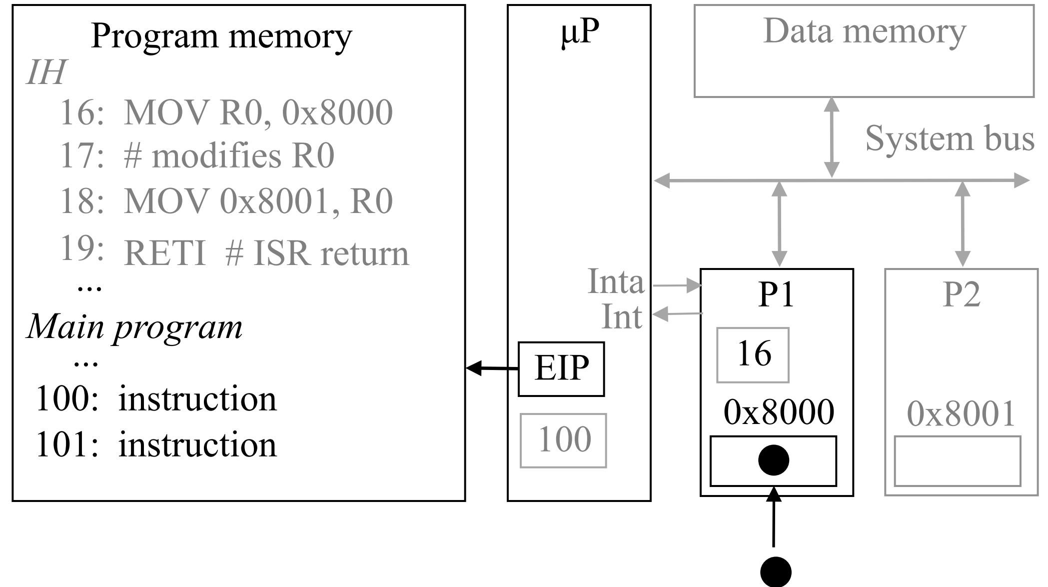
© Danilo Bruschi

Interrupt handling

(a): CPU sta eseguendo l'istruzione 100 di un programma

Concorrentemente

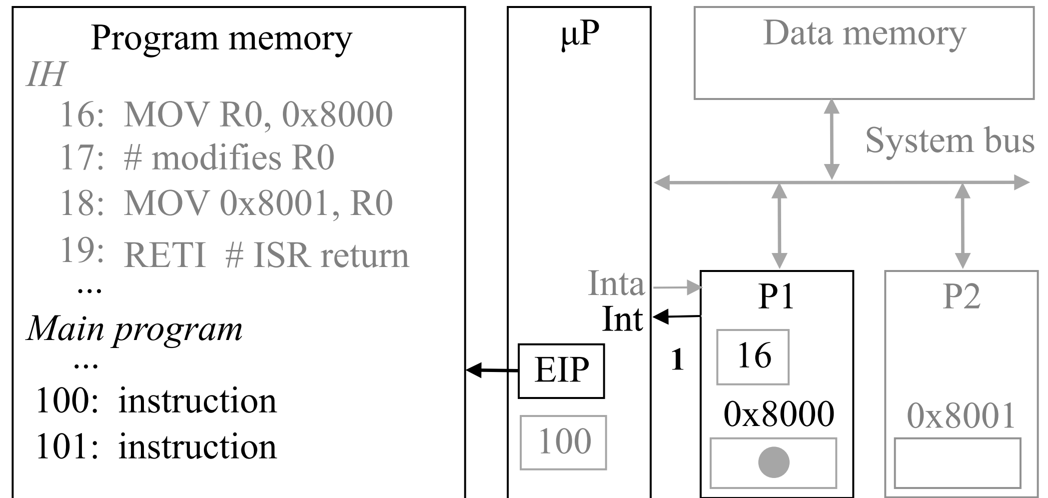
(b): P1 acquisisce dei dati in un suo registro di indirizzo 0x8000



© Danilo Bruschi

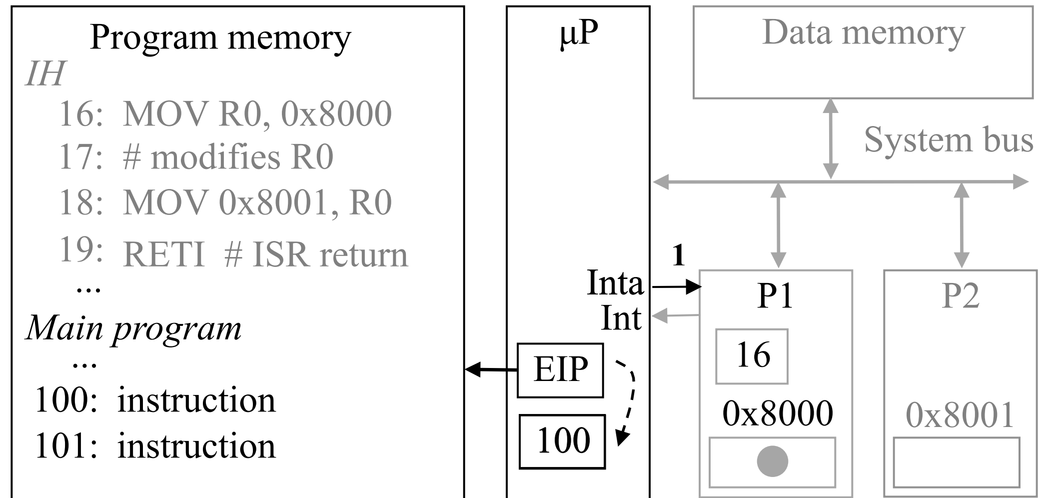
Interrupt handling

2: P1 asserisce il segnale *Int* per richiedere l'intervento del microprocessore



Interrupt handling

3: Dopo aver completato l'istruzione in esecuzione, il processore sente il segnale INT asserito, salva il valore di EIP e asserisce ***Inta***



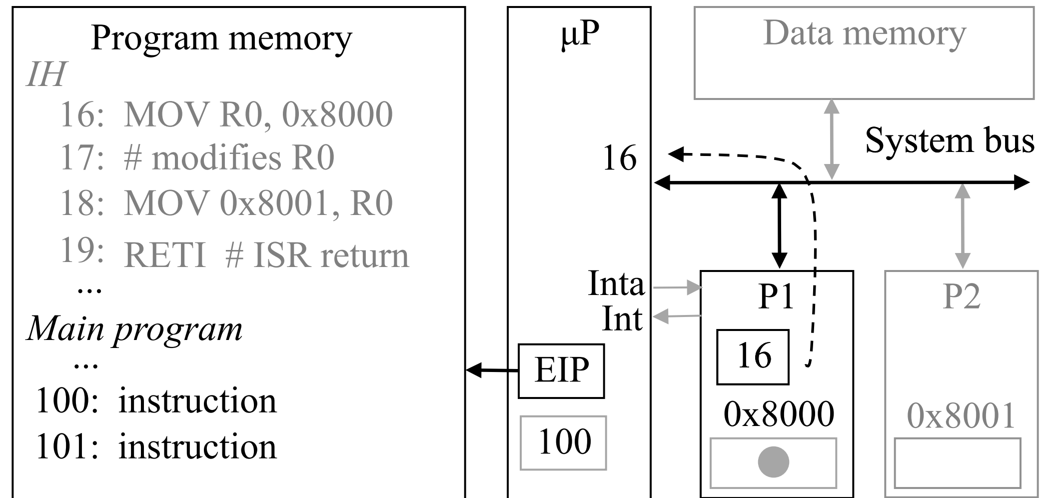
© Danilo Bruschi

Interrupt Handling

4(a): P1 rileva *Inta* e abbassa il segnale di Int

4(b): il processore riasserisce *Inta*

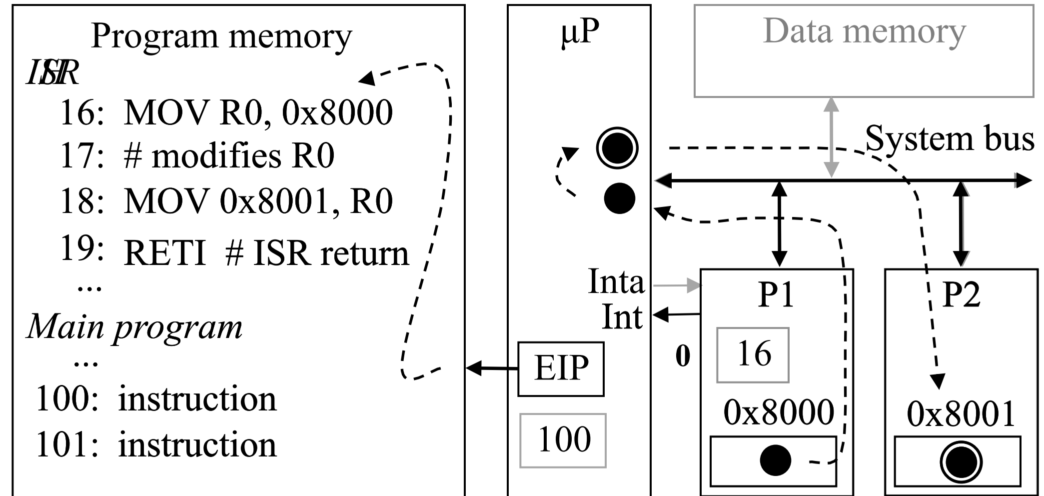
4(c): P1 rileva di nuovo Inta e pone il vettore dell'interrupt (**16**) sul bus dati



Interrupt handling

5(a): il processore salta all'interrupt handler associato all'interrupt 16.

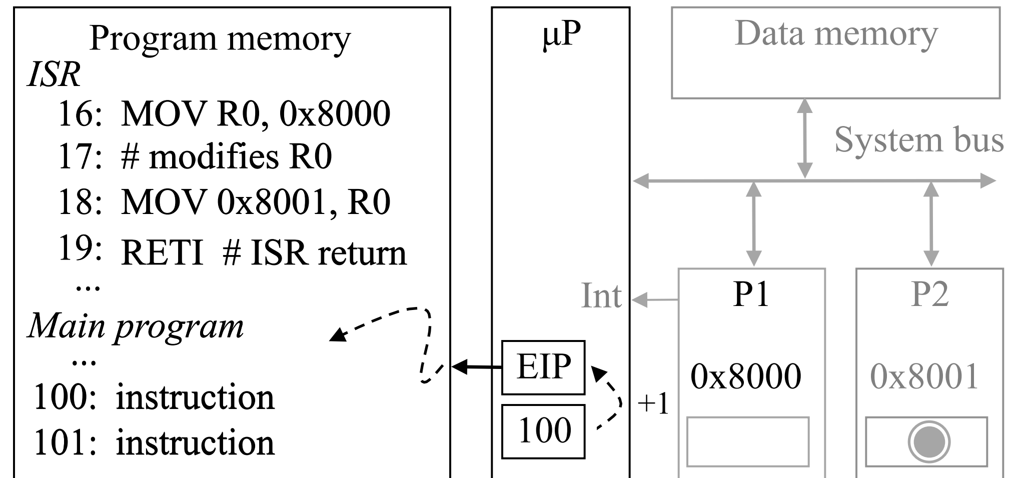
L'handler legge il dato da 0x8000, lo modifica e lo riscrive all'indirizzo 0x8001.



© Danilo Bruschi

Interrupt handling

6: Alla fine l'handler, esegue l'istruzione RETI che ripristina il valore di EIP a 100+1=101, da dove il processore riprende l'esecuzione



© Danilo Bruschi

Interrupt handling su IA-32

- Per svolgere le operazioni appena descritte l'architettura IA-32 prevede l'uso delle seguenti strutture dati:
  - Global Descriptor Table (GDT): definisce i contenuti dei diversi segmenti di memoria e informazioni per il controllo degli accessi
  - Interrupt Descriptor Table (IDT): definisce l'indirizzo d'inizio delle varie routine preposte alla gestione di eccezioni e interrupt
  - Task-State Segment (TSS): contiene gli indirizzi che devono essere caricati nel registri SS e ESP in risposta ad un'interruzione o interrupt e definiscono quindi uno stack di sistema usato nella fase di gestione di eccezioni e interrupt

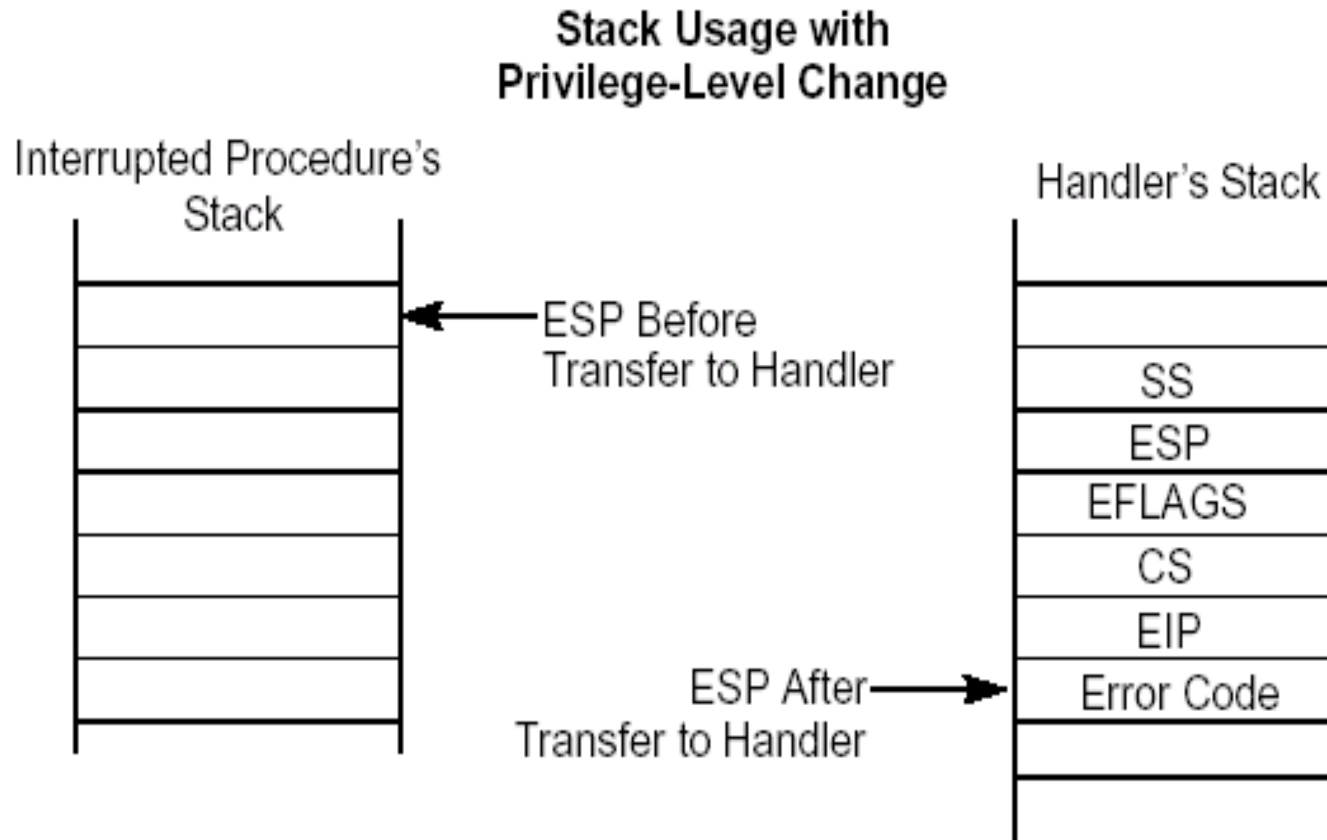
© Danilo Bruschi

Risposta a Interrupt ed Eccezioni

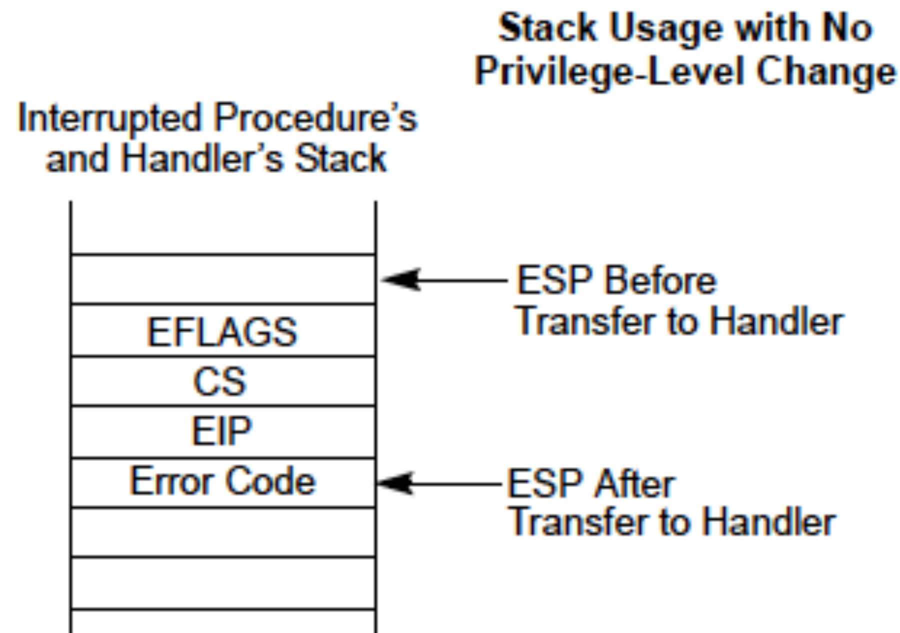
- Alla ricezione di un interrupt/eccezione l'hw risponde svolgendo le seguenti operazioni:
  - Recupera dal TSS del task in esecuzione, il selettore di segmento e il valore dello stack pointer per il nuovo stack
  - Memorizzare su questo nuovo stack:
    - lo stack segment selector (SS) del processo interrotto
    - lo stack pointer del programma interrotto (ESP)
    - EFLAGS, CS, e EIP correnti
    - Un eventuale errore code provocato da un'eccezione viene salvato sul nuovo stack dopo EIP
  - Carica nei registri SS e ESP i corrispondenti valori trovati in TSS
  - Carica in EIP l'indirizzo dell'handler che deve gestire l'interrupt/eccezione

© Danilo Bruschi

Configurazione stack

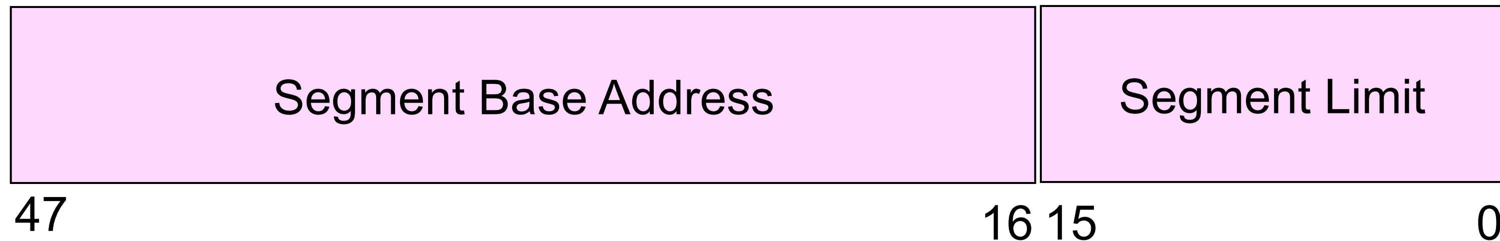


Configurazione Stack



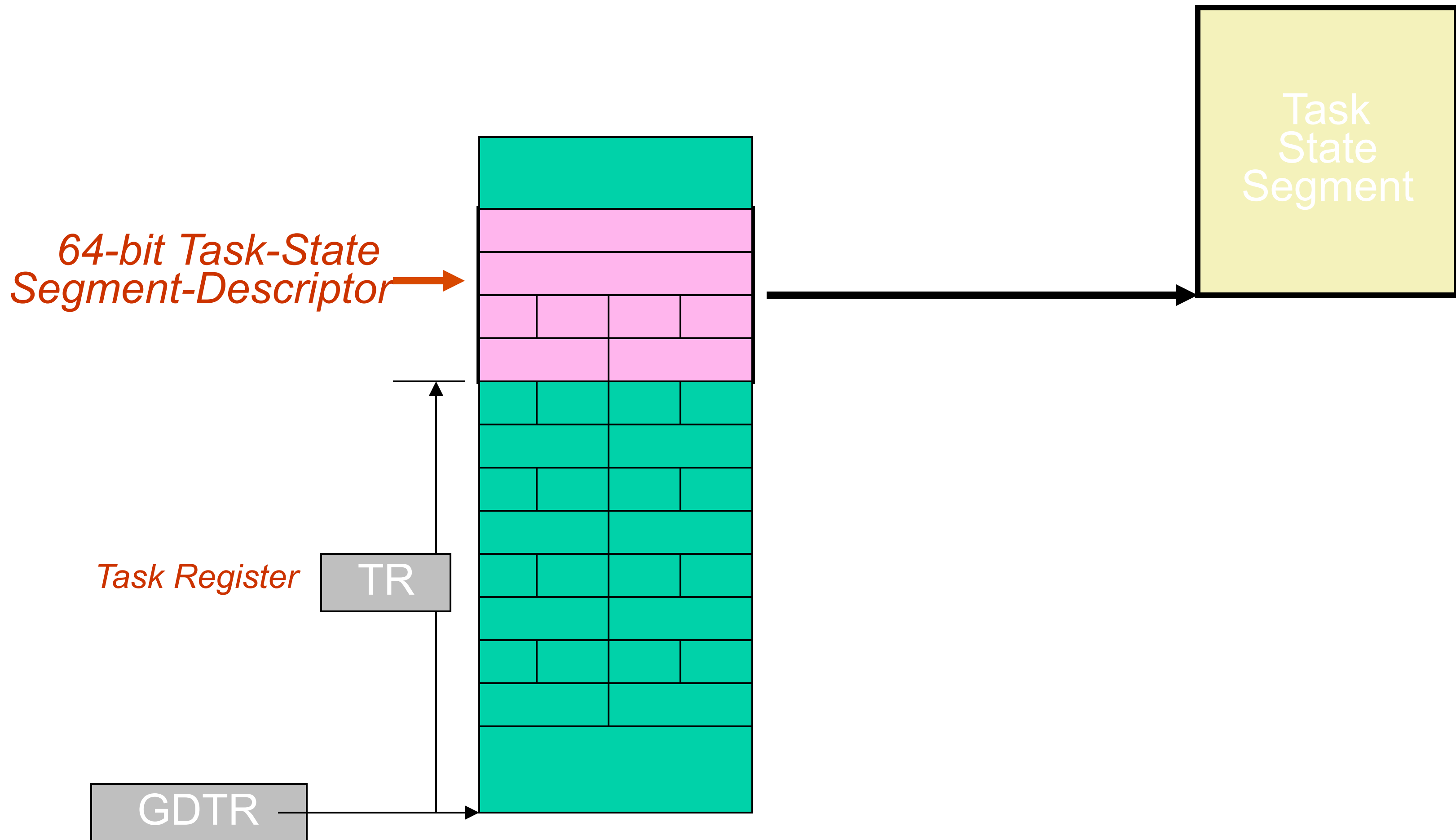
Come fa la CPU a localizzare GDT/IDT?

- Usa due registri dedicati a 48 bit: GDTR and IDTR
- Il valore di questi registri è caricato dal SO attraverso le istruzioni privilegiate: LGDT e LIDT
- I valori dei registri possono essere letti in user mode con le istruzioni: SGDT and SIDT



© Danilo Bruschi

How CPU finds the TSS



© Danilo Bruschi

Protected Control transfer

- Exceptions and interrupts are both "protected control transfers," which cause the processor to switch from user to kernel mode (CPL=0) without giving the user-mode code any opportunity to interfere with the functioning of the kernel or other environments
- In order to ensure that these protected control transfers are actually protected, the processor’s interrupt/exception mechanism is designed so that the code currently running when the interrupt or exception occurs does not get to choose arbitrarily where the kernel is entered or how

© Danilo Bruschi

Protected control transfer

- On the x86, two mechanisms work together to provide this protection:
  - The Interrupt Descriptor Table. The processor ensures that interrupts and exceptions can only cause the kernel to be entered at a few specific, well-defined entry-points determined by the kernel itself
  - The Task State Segment: this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel. For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory

TSS in JOS

- Although the TSS is large and can potentially serve a variety of purposes, JOS only uses it to define the kernel stack that the processor should switch to when it transfers from user to kernel mode
- Since "kernel mode" in JOS is privilege level 0 on the x86, the processor uses the ESP0 and SS0 fields of the TSS to define the kernel stack when entering kernel mode. JOS doesn't use any other TSS fields

TSS

```
// Task state segment format (as described by the Pentium architecture
book)
struct Taskstate {
    uint32_t ts_link;// Old ts selector
    uintptr_t ts_esp0;    // Stack pointers and segment selectors
    uint16_t ts_ss0; //    after an increase in privilege level
    uint16_t ts_padding1;
    uintptr_t ts_esp1;
    uint16_t ts_ss1;
    uint16_t ts_padding2;
    uintptr_t ts_esp2;
    uint16_t ts_ss2;
    uint16_t ts_padding3;
    physaddr_t ts_cr3;    // Page directory base
    uintptr_t ts_eip;// Saved state from last task switch
    uint32_t ts_eflags;
    uint32_t ts_eax; // More saved state (registers)
```


© Danilo Bruschi

```
    uint32_t ts_ecx;
    uint32_t ts_edx;
    uint32_t ts_ebx;
    uintptr_t ts_esp;
    uintptr_t ts_ebp;
    uint32_t ts_esi;
    uint32_t ts_edi;
    uint16_t ts_es;        // Even more saved state (segment selectors)
    uint16_t ts_padding4;
    uint16_t ts_cs;
    uint16_t ts_padding5;
    uint16_t ts_ss;
    uint16_t ts_padding6;
    uint16_t ts_ds;
    uint16_t ts_padding7;
    uint16_t ts_fs;
    uint16_t ts_padding8;
    uint16_t ts_gs;
    uint16_t ts_padding9;
    uint16_t ts_ldt;
    uint16_t ts_padding10;
    uint16_t ts_t;         // Trap on task switch
    uint16_t ts_iomb; // I/O map base address
};
```


© Danilo Bruschi

TSS

```c
// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
	// Setup a TSS so that we get the right stack
	// when we trap to the kernel.
	ts.ts_esp0 = KSTACKTOP;
	ts.ts_ss0 = GD_KD;
	// Initialize the TSS slot of the gdt.
	gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
					sizeof(struct Taskstate), 0);
	gdt[GD_TSS0 >> 3].sd_s = 0;

	// Load the TSS selector (like other segment selectors, the
	// bottom three bits are special; we leave them 0)
	ltr(GD_TSS0);
	// Load the IDT
	lidt(&idt_pd);
}
```

GDT

```
// Global descriptor numbers
#define GD_KT     0x08     // kernel text
#define GD_KD     0x10     // kernel data
#define GD_UT     0x18     // user text
#define GD_UD     0x20     // user data
#define GD_TSS0   0x28     // Task segment selector for CPU 0

/* Interrupt descriptor table.  (Must be built at run time because
 * shifted function addresses can't be represented in relocation records.)
 */
struct Gatedesc idt[256] = { { 0 } };
struct Pseudodesc idt_pd = {
        sizeof(idt) - 1, (uint32_t) idt
};
```


© Danilo Bruschi

SETTING UP IDT

© Danilo Bruschi

Example

Let's put these pieces together and trace through an example. Let's say the processor is executing code in a user environment and encounters a divide instruction that attempts to divide by zero.

1. The processor switches to the stack defined by the `SS0` and `ESP0` fields of the TSS, which in JOS will hold the values `GD_KD` and `KSTACKTOP`, respectively.
2. The processor pushes the exception parameters on the kernel stack, starting at address `KSTACKTOP`:

```
+--------------------+ KSTACKTOP
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20 <---- ESP
+--------------------+
```

3. Because we're handling a divide error, which is interrupt vector 0 on the x86, the processor reads IDT entry 0 and sets `CS:EIP` to point to the handler function described by the entry.
4. The handler function takes control and handles the exception, for example by terminating the user environment.

Exception

- All of the synchronous exceptions that the x86 processor can generate internally use interrupt vectors between 0 and 31, and therefore map to IDT entries 0-31
  - For example, a page fault always causes an exception through vector 14.
  - Interrupt vectors greater than 31 are only used by software interrupts, which can be generated by the `int` instruction, or asynchronous hardware interrupts caused by external devices when they need attention

© Danilo Bruschi

Interrupt in JOS

- External interrupts (i.e., device interrupts) are referred to as IRQs. There are 16 possible IRQs, numbered 0 through 15
- The mapping from IRQ number to IDT entry is not fixed. `pic_init` in `picirq.c` maps IRQs 0-15 to IDT entries starting from `IRQ_OFFSET` through `IRQ_OFFSET+15`
- In inc/trap.h, IRQ_OFFSET is defined to be decimal 32. Thus the IDT entries 32-47 correspond to the IRQs 0-15.
- For example, the clock interrupt is IRQ 0. Thus, IDT[IRQ_OFFSET+0] (i.e., IDT[32]) contains the address of the clock's interrupt handler routine in the kernel

© Danilo Bruschi

Interrupt in JOS

- In JOS, we make a key simplification: external device interrupts are *always* disabled when in the kernel
- External interrupts are controlled by the FL_IF flag bit of the `%eflags` register
- When this bit is set, external interrupts are enabled
- While the bit can be modified in several ways, because of our simplification, we will handle it solely through the process of saving and restoring `%eflags` register as we enter and leave user mode

© Danilo Bruschi

IDT

Table 6-1. Protected-Mode Exceptions and Interrupts

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.

© Danilo Bruschi

IDT

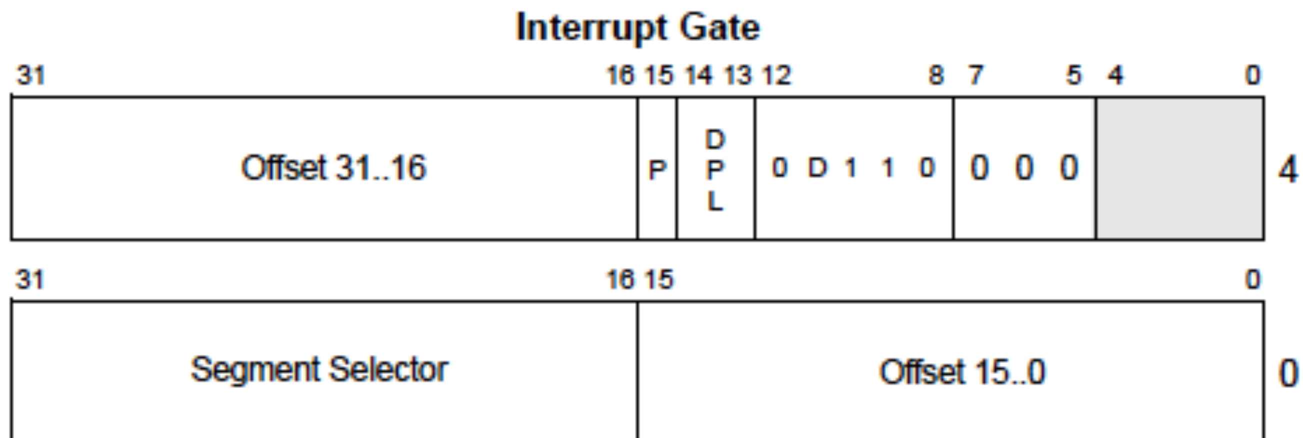
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]

Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)

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]
20-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

© Danilo Bruschi

Interrupt Gate



© Danilo Bruschi

inc/trap.h

```
// Trap numbers
// These are processor defined:
#define T_DIVIDE     0          // divide error
#define T_DEBUG      1          // debug exception
#define T_NMI        2          // non-maskable interrupt
#define T_BRKPT      3          // breakpoint
#define T_OFLOW      4          // overflow
#define T_BOUND      5          // bounds check
#define T_ILLOP      6          // illegal opcode
#define T_DEVICE     7          // device not available
#define T_DBLFLT     8          // double fault
/* #define T_COPROC  9 */ // reserved (not generated by recent
processors)
#define T_TSS       10          // invalid task switch segment
#define T_SEGNP     11          // segment not present
#define T_STACK     12          // stack exception
#define T_GPFLT     13          // general protection fault
#define T_PGFLT     14          // page fault
```

inc/trap.h

```
// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.

#define T_SYSCALL   48     // system call

#define T_DEFAULT   500       // catchall
```


© Danilo Bruschi

IRQ

- `#define IRQ_OFFSET    32  // IRQ 0 corresponds to int IRQ_OFFSET`
- `// Hardware IRQ numbers. We receive these as (IRQ_OFFSET +IRQ_WHATEVER)`
- `#define IRQ_TIMER        0`
- `#define IRQ_KBD          1`
- `#define IRQ_SERIAL       4`
- `#define IRQ_SPURIOUS     7`
- `#define IRQ_IDE         14`
- `#define IRQ_ERROR       19`

IDT

```
/* Interrupt descriptor table.  (Must be built at run time because
 * shifted function addresses can't be represented in relocation
records.)
 */
struct Gatedesc idt[256] = { { 0 } };
struct Pseudodesc idt_pd = {
        sizeof(idt) - 1, (uint32_t) idt
```


© Danilo Bruschi

Interrupt gate

```
// Gate descriptors for interrupts and traps
struct Gatedesc {
    unsigned gd_off_15_0 : 16;   // low 16 bits of offset in segment
    unsigned gd_ss : 16;        // segment selector
    unsigned gd_args : 5;        // # args, 0 for interrupt/trap gates
    unsigned gd_rsv1 : 3;        // reserved(should be zero I guess)
    unsigned gd_type : 4;        // type(STS_{TG,IG32,TG32})
    unsigned gd_s : 1;           // must be 0 (system)
    unsigned gd_dpl : 2;        // descriptor(meaning new) priv. level
    unsigned gd_p : 1;           // Present
    unsigned gd_off_31_16 : 16;  // high bits of offset in segment
};
```

Building the IDT

- In order to built the IDT we will perform the following steps:
  1. Programming the handling routine
  2. Inserting the address of the the handling routine as well as the remaining parameters inside the correct postion in the IDT

© Danilo Bruschi

PROGRAMMING THE HANDLING ROUTINE (TRAPENTRY.S)

© Danilo Bruschi

TRAPHANDLER

```
/* TRAPHANDLER defines a globally-visible function for handling a
* trap. It pushes a trap number onto the stack, then jumps to
*_alltraps. Use TRAPHANDLER for traps where the CPU automatically
* pushes an error code.
*
* You shouldn't call a TRAPHANDLER function from C, but you may
* need to _declare_ one in C (for instance, to get a function pointer
* during IDT setup).  You can declare the function with void NAME();
* where NAME is the argument passed to TRAPHANDLER.
*/
```


© Danilo Bruschi

Traphandler

```
#define TRAPHANDLER(name, num)                      \
    .globl name;      /* define global symbol for 'name' */\
    .type name, @function; /* symbol type is function */    \
    .align 2;      /* align function definition */      \
    name:          /* function starts here */    \
    pushl $(num);                                \
    jmp _alltraps

#define TRAPHANDLER_NOEC(name, num)                 \
    .globl name;                                 \
    .type name, @function;                           \
    .align 2;                                \
    name:                                    \
    pushl $0;                                \
    pushl $(num);                                \
    jmp _alltraps
```


© Danilo Bruschi

_alltraps

```
/*
 * Lab 3: Your code here for _alltraps
 */
_alltraps:
       pushl %ds
       pushl %es
       pusha
       movl $GD_KD, %eax
       movw %ax, %es
       movw %ax, %ds
       push %esp        // trap parameter
       call trap
       //NEVER RETURN HERE !!!!
```


© Danilo Bruschi

```
/*
 * generating entry points for the different traps.
 */
    TRAPHANDLER_NOEC (handler0, T_DIVIDE);
    TRAPHANDLER_NOEC (handler1, T_DEBUG);
    TRAPHANDLER_NOEC (handler2, T_NMI);
    TRAPHANDLER_NOEC (handler3, T_BRKPT);
    TRAPHANDLER_NOEC (handler4, T_OFLOW);
    TRAPHANDLER_NOEC (handler5, T_BOUND );
    TRAPHANDLER_NOEC (handler6, T_ILLOP);
    TRAPHANDLER_NOEC (handler7, T_DEVICE);
    TRAPHANDLER (handler8, T_DBLFLT);
    TRAPHANDLER (handler10, T_TSS);
    TRAPHANDLER (handler11, T_SEGNP);
    TRAPHANDLER (handler12, T_STACK);
    TRAPHANDLER (handler13, T_GPFLT);
    TRAPHANDLER (handler14, T_PGFLT);
    TRAPHANDLER_NOEC (handler16, T_FPERR);
    …
```

```
/* setup hardware interrupts */
    TRAPHANDLER_NOEC(irq0_entry, IRQ_OFFSET+0);
    TRAPHANDLER_NOEC(irq1_entry, IRQ_OFFSET+1);
    TRAPHANDLER_NOEC(irq2_entry, IRQ_OFFSET+2);
    TRAPHANDLER_NOEC(irq3_entry, IRQ_OFFSET+3);
    TRAPHANDLER_NOEC(irq4_entry, IRQ_OFFSET+4);
    TRAPHANDLER_NOEC(irq5_entry, IRQ_OFFSET+5);
    TRAPHANDLER_NOEC(irq6_entry, IRQ_OFFSET+6);
    TRAPHANDLER_NOEC(irq7_entry, IRQ_OFFSET+7);
    TRAPHANDLER_NOEC(irq8_entry, IRQ_OFFSET+8);
    TRAPHANDLER_NOEC(irq9_entry, IRQ_OFFSET+9);
    TRAPHANDLER_NOEC(irq10_entry, IRQ_OFFSET+10);
    TRAPHANDLER_NOEC(irq11_entry, IRQ_OFFSET+11);
    TRAPHANDLER_NOEC(irq12_entry, IRQ_OFFSET+12);
    TRAPHANDLER_NOEC(irq13_entry, IRQ_OFFSET+13);
    TRAPHANDLER_NOEC(irq14_entry, IRQ_OFFSET+14);
/*
```


© Danilo Bruschi

IDT INITIALIZATION

```
void
trap_init(void)
{   extern struct Segdesc gdt[];
    extern void handler0();
    extern void handler1();
    extern void handler2();
    extern void handler3();
    extern void handler4();
    extern void handler5();
    extern void handler6();
    extern void handler7();
    extern void handler8();
    extern void handler10();
    extern void handler11();
    extern void handler12();
    extern void handler13();
    extern void handler14();
    …
```


© Danilo Bruschi

```
// hardware interrupts
    extern void irq0_entry();
    extern void irq1_entry();
    extern void irq2_entry();
    extern void irq3_entry();
    extern void irq4_entry();
    extern void irq5_entry();
    extern void irq6_entry();
    extern void irq7_entry();
    extern void irq8_entry();
    extern void irq9_entry();
    extern void irq10_entry();
    extern void irq11_entry();
    extern void irq12_entry();
    extern void irq13_entry();
    extern void irq14_entry();
```

```
SETGATE (idt[T_DIVIDE], 0, GD_KT, handler0, 0);
SETGATE (idt[T_DEBUG], 0, GD_KT, handler1, 0);
SETGATE (idt[T_NMI], 0, GD_KT, handler2, 0);
SETGATE (idt[T_BRKPT], 0, GD_KT, handler3, 3);
SETGATE (idt[T_OFLOW], 0, GD_KT, handler4, 0);
SETGATE (idt[T_BOUND], 0, GD_KT, handler5, 0);
SETGATE (idt[T_ILLOP], 0, GD_KT, handler6, 0);
SETGATE (idt[T_DEVICE], 0, GD_KT, handler7, 0);
SETGATE (idt[T_DBLFLT], 0, GD_KT, handler8, 0);
SETGATE (idt[T_TSS], 0, GD_KT, handler10, 0)
SETGATE (idt[T_SEGNP], 0, GD_KT, handler11, 0);
SETGATE (idt[T_STACK], 0, GD_KT, handler12, 0);
SETGATE (idt[T_GPFLT], 0, GD_KT, handler13, 0);
SETGATE (idt[T_PGFLT], 0, GD_KT, handler14, 0);
SETGATE (idt[T_FPERR], 0, GD_KT, handler16, 0);
```

inc/mmu.h

```
#define SETGATE(gate, istrap, sel, off, dpl)                    \
{                                                               \
     (gate).gd_off_15_0 = (uint32_t) (off) & 0xffff;            \
     (gate).gd_ss = (sel);                                      \
     (gate).gd_args = 0;                                        \
     (gate).gd_rsv1 = 0;                                        \
     (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;           \
     (gate).gd_s = 0;                                           \
     (gate).gd_dpl = (dpl);                                     \
     (gate).gd_p = 1;                                           \
     (gate).gd_off_31_16 = (uint32_t) (off) >> 16;              \
}
```

inc/mmu.h

```
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   an interrupt gate and a trap gate is in the effect on IF (the
//   interrupt-enable flag). An interrupt that vectors through an
//   interrupt gate resets IF, thereby preventing other interrupts
from
//   interfering with the current interrupt handler. A subsequent IRET
//   instruction restores IF to the value in the EFLAGS image on the
//   stack. An interrupt through a trap gate does not change IF."
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//    the privilege level required for software to invoke
//    this interrupt/trap gate explicitly using an int instruction.
```


© Danilo Bruschi

trap_init_percpu

```
thiscpu->cpu_ts.ts_esp0 = KSTACKTOP -
cpunum()*(KSTKSIZE+KSTKGAP);
   thiscpu->cpu_ts.ts_ss0 = GD_KD;
// Initialize the TSS slot of the gdt.
   gdt[(GD_TSS0 >> 3) + cpunum()] = SEG16(STS_T32A,
                (uint32_t) (&thiscpu->cpu_ts),
               sizeof(struct Taskstate), 0);
   gdt[(GD_TSS0 >> 3) + cpunum()].sd_s = 0;
// Load the TSS selector (like other segment selectors,
the bottom three bits are special; we leave them 0)
   ltr(GD_TSS0 + (cpunum()<< 3)) ;
// Load the IDT
   lidt(&idt_pd);
```

EVENT HANDLER

Trapframe

```
struct Trapframe {
     struct PushRegs tf_regs;
     uint16_t tf_es;
     uint16_t tf_padding1;
     uint16_t tf_ds;
     uint16_t tf_padding2;
     uint32_t tf_trapno;
     /* below here defined by x86 hardware */
     uint32_t tf_err;
     uintptr_t tf_eip;
     uint16_t tf_cs;
     uint16_t tf_padding3;
     uint32_t tf_eflags;
     /* below here only when crossing rings, such as from user to kernel */
     uintptr_t tf_esp;
     uint16_t tf_ss;
     uint16_t tf_padding4;
} __attribute__((packed));
```

inc/trap.h

```
struct PushRegs {
	/* registers as pushed by pusha */
	uint32_t reg_edi;
	uint32_t reg_esi;
	uint32_t reg_ebp;
	uint32_t reg_oesp;		/* Useless */
	uint32_t reg_ebx;
	uint32_t reg_edx;
	uint32_t reg_ecx;
	uint32_t reg_eax;
};
```


© Danilo Bruschi

kern/trap.c

```
void
trap(struct Trapframe *tf)
{
    if ((tf->tf_cs & 3) == 3) {
        // Trapped from user mode.
        // Copy trap frame (which is currently on the stack)
        // into 'curenv->env_tf', so that running the environment
        // will restart at the trap point.
        assert(curenv);
        curenv->env_tf = *tf;
        // The trapframe on the stack should be ignored from here on.
        tf = &curenv->env_tf;
    }
    // Dispatch based on what type of trap occurred
    trap_dispatch(tf);

…
```

Clock driver

```
static void
trap_dispatch(struct Trapframe *tf)
{
// Handle processor exceptions/interrupts

// Handle clock interrupts. Don't forget to acknowledge the
// interrupt using lapic_eoi() before calling the scheduler!

    if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
            lapic_eoi();
            sched_yield();
            return;
        }
```


© Danilo Bruschi

Keyborad Driver

```
// Handle keyboard and serial interrupts.
    // LAB 5: Your code here.
    if (tf->tf_trapno == IRQ_OFFSET + IRQ_KBD) {
        kbd_intr();
        return;
        }
    if (tf->tf_trapno == IRQ_OFFSET + IRQ_SERIAL) {
        serial_intr();
        return;
        }
```


© Danilo Bruschi

trap_dispatch 2

```
if (tf->tf_cs == GD_KT){
        print_trapframe(tf);
        panic("unhandled trap in kernel");
        }
switch (tf->tf_trapno) {
    case T_PGFLT:
        page_fault_handler (tf);
        break;
    case T_BRKPT:
        monitor(tf);
        break;
```


© Danilo Bruschi

trap_dispatch 3

```
case T_SYSCALL:
      tf->tf_regs.reg_eax =
      syscall(tf->tf_regs.reg_eax,
            tf->tf_regs.reg_edx,
            tf->tf_regs.reg_ecx,
            tf->tf_regs.reg_ebx,
            tf->tf_regs.reg_edi,
            tf->tf_regs.reg_esi);
         return;
      default:
      env_destroy(curenv); //set to NULL curenv
      return;
}
```


© Danilo Bruschi

trap

```
// If we made it to this point, then no other environment was
// scheduled, so we should return to the current environment
// if doing so makes sense.

    if (curenv && curenv->env_status == ENV_RUNNING)
        env_run(curenv);
    else
        sched_yield();
}
```

SCHEDULER & CONTEXT SWITCH

© Danilo Bruschi

JOS PCB (ENV)

```
struct Env {
    struct Trapframe env_tf;     // Saved registers
    struct Env *env_link;        // Next free Env
    envid_t env_id;              // Unique environment identifier
    envid_t env_parent_id;       // env_id of this env's parent
    enum EnvType env_type;       // Indicates special system enviro
    unsigned env_status;         // Status of the environment
    uint32_t env_runs;           // Number of times environment has
    int env_cpunum;              // The CPU that the env is running on

    // Address space
    pde_t *env_pgdir;            // Kernel virtual address of page dir

    // Exception handling
    void *env_pgfault_upcall;    // Page fault upcall entry point

    // Lab 4 IPC
    bool env_ipc_recving;        // Env is blocked receiving
    void *env_ipc_dstva;         // VA at which to map received page
    uint32_t env_ipc_value;      // Data value sent to us
    envid_t env_ipc_from;        // envid of the sender
    int env_ipc_perm;            // Perm of page mapping received
};
```

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

scheduler

```
// Choose a user environment to run and run it.
void
sched_yield(void)
{
struct Env *idle;
int i,k ;
if (thiscpu->cpu_env == NULL) i = 0;
    else
    i = ENVX(thiscpu->cpu_env->env_id)+1;
for (k=0; k< NENV; k++) {
  if (envs[i].env_status == ENV_RUNNABLE)
       env_run (&envs[i]);
     i = (i+1)%NENV;
}
If ((thiscpu->cpu_env != NULL) && (thiscpu->cpu_env-
                                        >env_status == ENV_RUNNING))
    env_run(thiscpu->cpu_env);
```


© Danilo Bruschi

env_run in kern/env.c

```
// Context switch from curenv to env e.
// Note: if this is the first call to env_run, curenv is NULL.
//  (This function does not return.)
//
Void env_run(struct Env *e)
{
// Step 1: If this is a context switch (a new environment is running),
//          then set 'curenv' to the new environment,
//          update its 'env_runs' counter, and
//          and use lcr3() to switch to its address space.
```

kern/env.c

```
 if(curenv != e) {
        if (curenv && curenv->env_status == ENV_RUNNING)
            curenv->env_status = ENV_RUNNABLE;
            curenv = e;
            e->env_status = ENV_RUNNING;
            e->env_runs++;
           lcr3(e->env_cr3);
}
// Step 2: Use env_pop_tf() to restore the environment's
//          registers and drop into user mode in the
//          environment.

        env_pop_tf(&(e->env_tf));
```


© Danilo Bruschi

env_popf in kern/env.c

```
//
// Restores the register values in the Trapframe with the 'iret' instruction.
// This exits the kernel and starts executing some environment's code.
// This function does not return.
//
void
env_pop_tf(struct Trapframe *tf)
{
        __asm __volatile("movl %0,%%esp\n"
                "\tpopal\n"
                "\tpopl %%es\n"
                "\tpopl %%ds\n"
                "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
                "\tiret"
                : : "g" (tf) : "memory");
        panic("iret failed");  /* mostly to placate the compiler */
}
```

Idle

```c
// idle loop
#include <inc/x86.h>
#include <inc/lib.h>
void
umain(int argc, char **argv)
{
	binaryname = "idle";

	// Loop forever, simply trying to yield to a different environment.
	// Instead of busy-waiting like this,
	// a better way would be to use the processor's HLT instruction
	// to cause the processor to stop executing until the next interrupt -
	// doing so allows the processor to conserve power more effectively.
	while (1) {
		sys_yield();
	}
}
```


© Danilo Bruschi

SYSCALL

© Danilo Bruschi

`syscalls`

- When the user process invokes a system call, the processor enters kernel mode, the processor and the kernel cooperate to save the user process's state, the kernel executes appropriate code in order to carry out the system call, and then resumes the user process
- In the JOS kernel, we will use the `int` instruction, which causes a processor interrupt. We will use `int $0x30` as the system call interrupt

© Danilo Bruschi

syscalls

- The application will pass the system call number and the system call arguments in registers. This way, the kernel won't need to grub around in the user environment's stack or instruction stream.
- The system call number will go in **%eax**, and the arguments (up to five of them) will go in **%edx**, **%ecx**, **%ebx**, **%edi**, and **%esi**, respectively
- The assembly code to invoke a system call has been written for you, in syscall() in lib/syscall.c

© Danilo Bruschi

```
void
forktree(const char *cur)
{
   cprintf("%04x: I am '%s'\n", sys_getenvid(), cur);

   forkchild(cur, '0');
   forkchild(cur, '1');
}
```

“libc”

```
int
sys_cgetc(void)
{
        return syscall(SYS_cgetc, 0, 0, 0, 0, 0, 0);
}

int
sys_env_destroy(envid_t envid)
{
        return syscall(SYS_env_destroy, 1, envid, 0, 0, 0, 0);
}

envid_t
sys_getenvid(void)
{
         return syscall(SYS_getenvid, 0, 0, 0, 0, 0, 0);
}
```

```
/* !JOS_INC_SYSCALL_H */
/* system call numbers */
enum {
    SYS_cputs = 0,
    SYS_cgetc,
    SYS_getenvid,
    SYS_env_destroy,
    SYS_page_alloc,
    SYS_page_map,
    SYS_page_unmap,
    SYS_exofork,
    SYS_env_set_status,
    SYS_env_set_trapframe,
    SYS_env_set_pgfault_upcall,
    SYS_yield,
    SYS_ipc_try_send,
    SYS_ipc_recv,
    NSYSCALLS
};
```

lib/syscall.c

```
static inline int32_t
syscall(int num, int check, uint32_t a1, uint32_t a2, uint32_t
a3, uint32_t a4, uint32_t a5)
{  int32_t ret;
// Generic system call: pass system call number in AX,
// up to five parameters in DX, CX, BX, DI, SI.
// Interrupt kernel with T_SYSCALL.
//
// The "volatile" tells the assembler not to optimize
// this instruction away just because we don't use the
// return value.
//
// The last clause tells the assembler that this can
// potentially change the condition codes and arbitrary
// memory locations.
```

lib/syscall.c

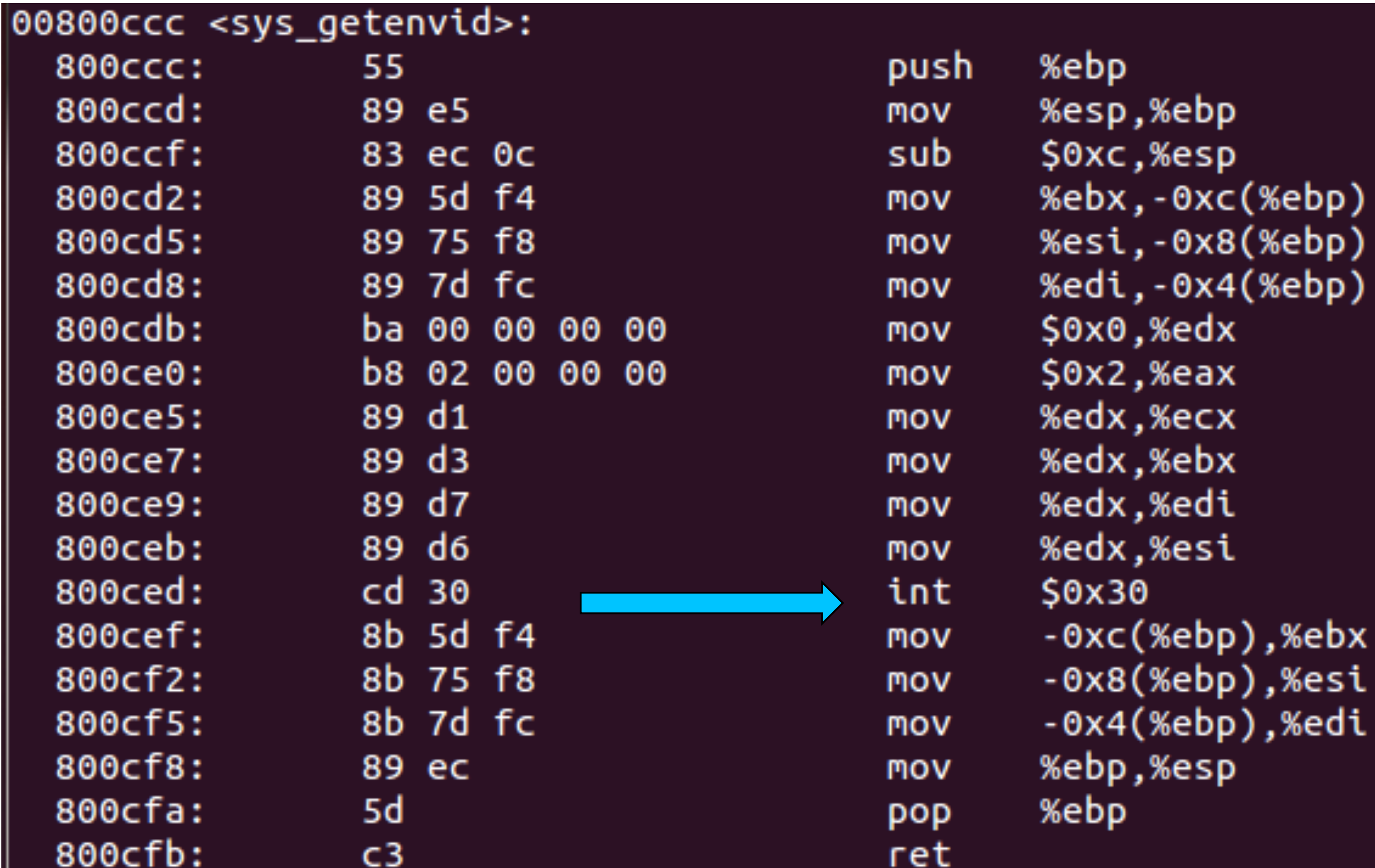
```
asm volatile("int %1\n"
        : "=a" (ret)
        : "i" (T_SYSCALL),
          "a" (num),
          "d" (a1),
          "c" (a2),
          "b" (a3),
          "D" (a4),
          "S" (a5)
        : "cc", "memory");

    if(check && ret > 0)
        panic("syscall %d returned %d (> 0)", num, ret);

    return ret;
}
```

```
mov num,%eax
mov a1,%edx
mov a2,%ecx
mov a3,%ebx
mov a4,%edi
mov a5,%esi
int T_SYSCALL
mov %eax,ret
```

objdump

```
00800034 <forktree>:
  800034:       55                      push   %ebp
  800035:       89 e5                   mov    %esp,%ebp
  800037:       53                      push   %ebx
  800038:       83 ec 14                sub    $0x14,%esp
  80003b:       8b 5d 08                mov    0x8(%ebp),%ebx
  80003e:       e8 89 0c 00 00          call   800ccc <sys_getenvid>
  800043:       89 5c 24 08             mov    %ebx,0x8(%esp)
  800047:       89 44 24 04             mov    %eax,0x4(%esp)
  80004b:       c7 04 24 e0 26 80 00    movl   $0x8026e0,(%esp)
  800052:       e8 b4 01 00 00          call   80020b <cprintf>
  800057:       c7 44 24 04 30 00 00    movl   $0x30,0x4(%esp)
  80005e:       00
  80005f:       89 1c 24                mov    %ebx,(%esp)
  800062:       e8 16 00 00 00          call   80007d <forkchild>
  800067:       c7 44 24 04 31 00 00    movl   $0x31,0x4(%esp)
  80006e:       00
```

sys_yield



kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
        // Handle processor exceptions.
      switch (tf->tf_trapno) {
        case T_PGFLT:
                page_fault_handler(tf);
                return;
        case T_BRKPT:
                monitor(tf);
                return;
        case T_SYSCALL:
                tf->tf_regs.reg_eax =
                syscall(tf->tf_regs.reg_eax,
                        tf->tf_regs.reg_edx,
                        tf->tf_regs.reg_ecx,
                        tf->tf_regs.reg_ebx,
                        tf->tf_regs.reg_edi,
                        tf->tf_regs.reg_esi);
                return;
        ….
        }
```

KERN/SYSCALL.C

```
// Dispatches to the correct kernel function, passing the arguments.

int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3,
        uint32_t a4, uint32_t a5)
{
// Call the function corresponding to the 'syscallno'
// Return any appropriate return value.

switch (syscallno) {
    case SYS_cputs:
        sys_cputs((char *) a1, a2);
        return 0;
    case SYS_cgetc:
        return sys_cgetc();
    case SYS_getenvid:
        return sys_getenvid();
    case SYS_env_destroy:
        return sys_env_destroy(a1);
    case SYS_yield:
         sys_yield();
         return 0;
```

sys_getenvid in kern/syscall.c

```
// Returns the current environment's envid.
static envid_t
sys_getenvid(void)
{
   return curenv->env_id;
}
```


© Danilo Bruschi

THE WAY BACK

© Danilo Bruschi

KERN/SYSCALL.C

```
// Dispatches to the correct kernel function, passing the
arguments.
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t
a3, uint32_t a4, uint32_t a5)
{
    // Call the function corresponding to the 'syscallno'
    // Return any appropriate return value.

    switch (syscallno) {
    case SYS_cputs:
        sys_cputs((char *) a1, a2);
        return 0;
    case SYS_cgetc:
        return sys_cgetc();
    case SYS_getenvid:
        return sys_getenvid();
    case SYS_env_destroy:
        return sys_env_destroy(a1);
    case SYS_yield:
         sys_yield();
         return 0;
```

kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
        // Handle processor exceptions.
      switch (tf->tf_trapno) {
        case T_PGFLT:
                page_fault_handler(tf);
                return;
        case T_BRKPT:
                monitor(tf);
                return;
        case T_SYSCALL:
                tf->tf_regs.reg_eax =
                syscall(tf->tf_regs.reg_eax,
                        tf->tf_regs.reg_edx,
                        tf->tf_regs.reg_ecx,
                        tf->tf_regs.reg_ebx,
                        tf->tf_regs.reg_edi,
                        tf->tf_regs.reg_esi);
                return;
        ….
        }
```


© Danilo Bruschi

trap

```
// If we made it to this point, then no other environment was
// scheduled, so we should return to the current environment
// if doing so makes sense.

    if (curenv && curenv->env_status == ENV_RUNNING)
        env_run(curenv);
    else
        sched_yield();
}
```


© Danilo Bruschi

env_run in kern/env.c

```
// Context switch from curenv to env e.
// Note: if this is the first call to env_run, curenv is NULL.
//  (This function does not return.)
//
Void env_run(struct Env *e)
{
// Step 1: If this is a context switch (a new environment is running),
//         then set 'curenv' to the new environment,
//         update its 'env_runs' counter, and
//         and use lcr3() to switch to its address space.
```

kern/env.c

```
 if(curenv != e) {
        if (curenv && curenv->env_status == ENV_RUNNING)
            curenv->env_status = ENV_RUNNABLE;
            curenv = e;
            e->env_status = ENV_RUNNING;
            e->env_runs++;
           lcr3(e->env_cr3);
}
// Step 2: Use env_pop_tf() to restore the environment's
//         registers and drop into user mode in the
//         environment.

        env_pop_tf(&(e->env_tf));
```

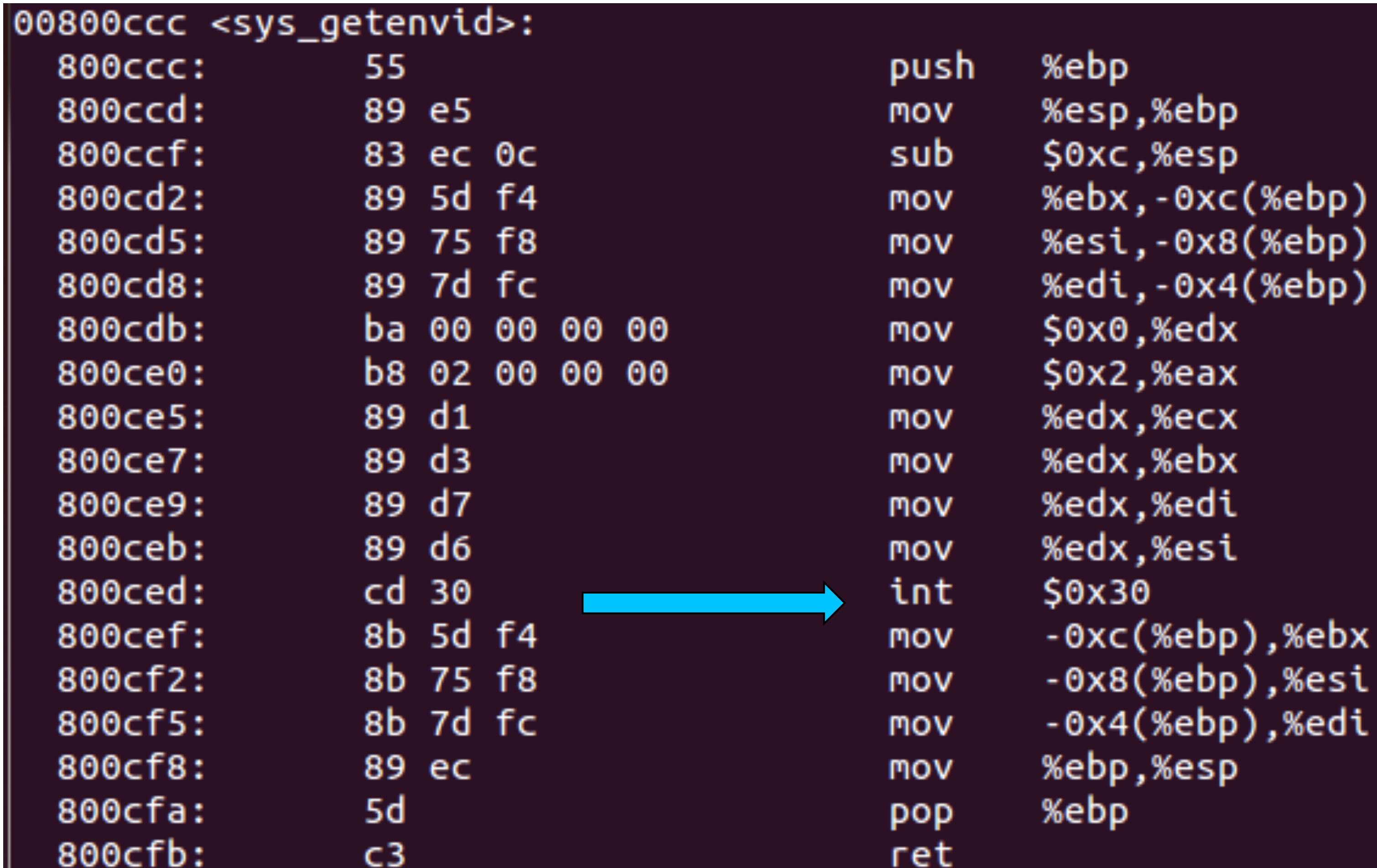

© Danilo Bruschi

env_popf in kern/env.c

```
//
// Restores the register values in the Trapframe with the 'iret' instruction.
// This exits the kernel and starts executing some environment's code.
// This function does not return.
//
void
env_pop_tf(struct Trapframe *tf)
{
        __asm __volatile("movl %0,%%esp\n"
                "\tpopal\n"
                "\tpopl %%es\n"
                "\tpopl %%ds\n"
                "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
                "\tiret"
                : : "g" (tf) : "memory");
        panic("iret failed");  /* mostly to placate the compiler */
}
```


© Danilo Bruschi

sys_yield



Objdump: call libc

```
00800034 <forktree>:
  800034:       55                      push   %ebp
  800035:       89 e5                   mov    %esp,%ebp
  800037:       53                      push   %ebx
  800038:       83 ec 14                sub    $0x14,%esp
  80003b:       8b 5d 08                mov    0x8(%ebp),%ebx
  80003e:       e8 89 0c 00 00          call   800ccc <sys_getenvid>
  800043:       89 5c 24 08             mov    %ebx,0x8(%esp)
  800047:       89 44 24 04             mov    %eax,0x4(%esp)
  80004b:       c7 04 24 e0 26 80 00    movl   $0x8026e0,(%esp)
  800052:       e8 b4 01 00 00          call   80020b <cprintf>
  800057:       c7 44 24 04 30 00 00    movl   $0x30,0x4(%esp)
  80005e:       00
  80005f:       89 1c 24                mov    %ebx,(%esp)
  800062:       e8 16 00 00 00          call   80007d <forkchild>
  800067:       c7 44 24 04 31 00 00    movl   $0x31,0x4(%esp)
  80006e:       00
```

In conclusione

```
void
i386_init(void)
{   extern char edata[], end[];
    // Before doing anything else, complete the ELF loading process.
    // Clear the uninitialized global data (BSS) section of our program.
    // This ensures that all static/global variables start out zero.
    memset(edata, 0, end - edata);

    // Initialize the console.
    cons_init();
    cprintf("6828 decimal is %o octal!\n", 6828);

    // Lab 2 memory management initialization functions
    mem_init();

    // Lab 3 user environment initialization functions
    env_init();
    trap_init();
```

```
// Lab 4 multiprocessor initialization functions
     mp_init();
     lapic_init();
// Lab 4 multitasking initialization functions
     pic_init();
// Acquire the big kernel lock before waking up Aps
// Your code here:
     lock_kernel();
// Starting non-boot CPUs
     boot_aps();
// Start fs.
     ENV_CREATE(fs_fs, ENV_TYPE_FS);
     ENV_CREATE(user_icode, ENV_TYPE_USER);

// Should not be necessary - drains keyboard because interrupt has given up.
     kbd_intr();

// Schedule and run the first user environment!
     sched_yield();
}
```

SISTEMI OPERATIVI

FINE !!!