INTERRUPCIONES SOFTWARE y EXCEPCIONES

S.CANDELA

Interrupciones

Una interrupción se genera cuando se quiere que la CPU deje de ejecutar el proceso en curso y ejecute una función específica de quien produce la interrupción. Cuando se ejecuta esta función específica decimos que la CPU está atendiendo la interrupción. Podemos realizar una clasificación de las interrupciones, atendiendo a la fuente que las produce.

- **Interrupcion software**, se produce cuando un usuario solicita una llamada del sistema.
- **Interrupciones hardware**, son causadas cuando un dispositivo hardware requiere la atención de la CPU para que se ejecute su manejador.
- **Excepciones**, son interrupciones causadas por la propia CPU, cuando ocurre algún suceso, por ejemplo una división por cero.

Interrupción software (1)

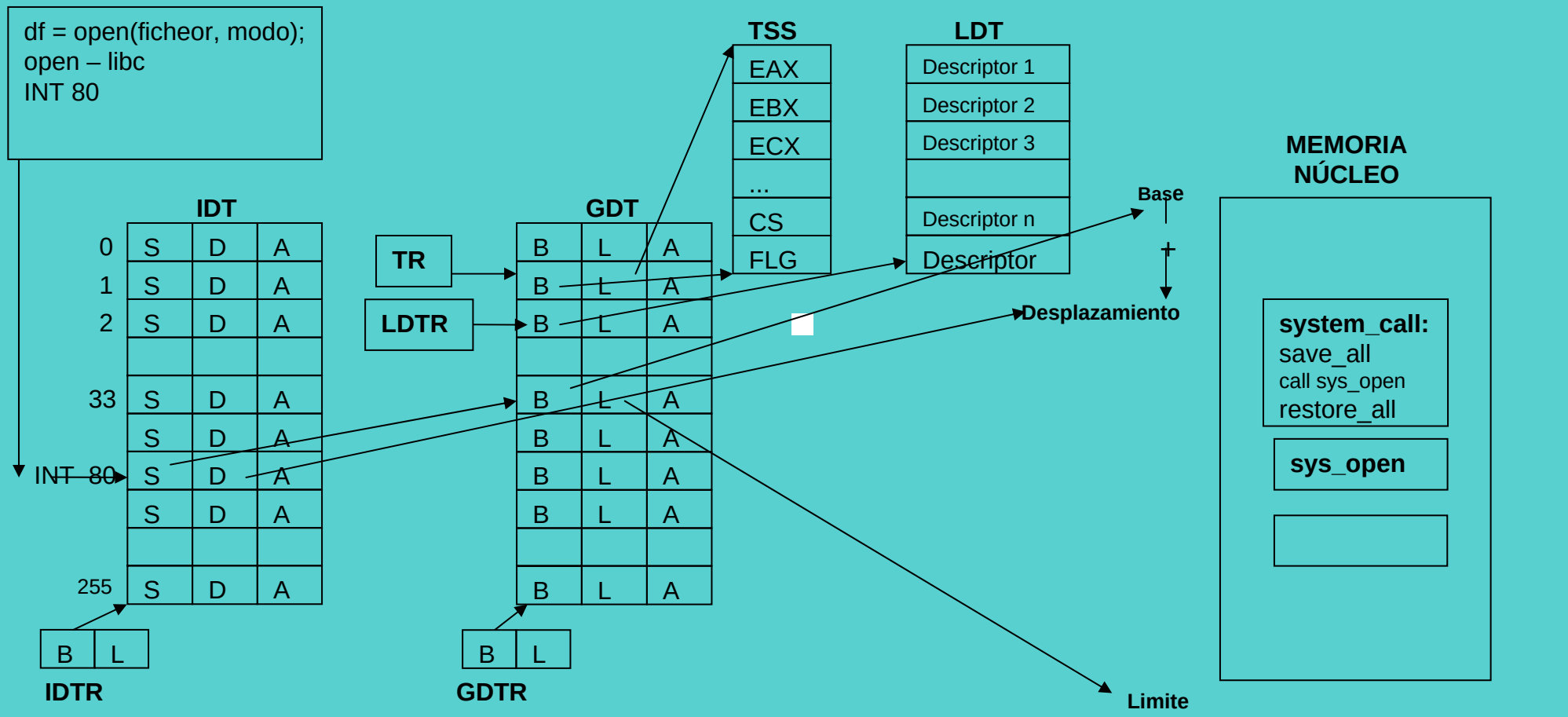
- Una Interrupción software se produce cuando un usuario solicita un recurso del núcleo, mediante una llamada al sistema, open, write, read, mount,..
- El proceso usuario solicita la función correspondiente de la librería libc.

  df = open (fichero, modo);
- La función de librería coloca los parámetros de la llamada en los registros del procesador y ejecuta la instrucción **INT 0x80**
- Se conmuta a modo núcleo mediante las tablas IDT y GDT
- Entra a ejecutarse una función del núcleo, **system_call**, Interfase entre el usuario y el núcleo.
- Cuando se termina la llamada, system_call retorna al proceso que la llamo y se retorna a modo usuario.

Interrupción software (2)

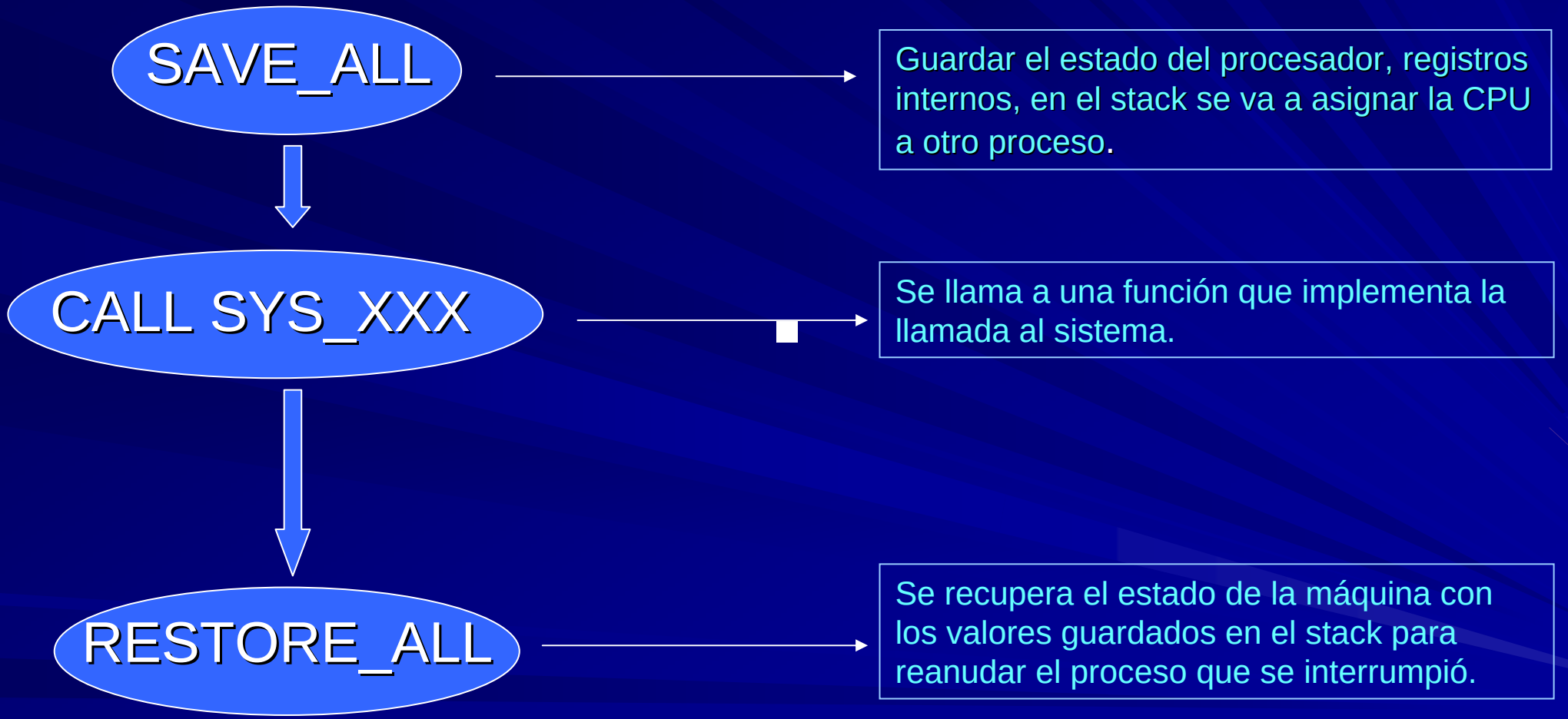
- Llega la interrupción software INT 80h y se busca en la IDT la entrada correspondiente a la interrupción.
- Se obtiene un puntero desplazamiento al núcleo (D)
- Y una entrada (S) a GDT
- La entrada en la GDT indica una base (B) y un limite (L) del núcleo
- Se llega al manejador de interrupción system_call.

Interrupciones Software gráfico



S – Selector
D – Desplazamiento
A – Atributos
B – Base
L - Limite

sys_call


SAVE_ALL
Guardar el estado del procesador, registros internos, en el stack se va a asignar la CPU a otro proceso.
CALL SYS_XXX
Se llama a una función que implementa la llamada al sistema.
RESTORE_ALL
Se recupera el estado de la máquina con los valores guardados en el stack para reanudar el proceso que se interrumpió.

System_call

ENTRY(system_call)
/* el primer argumento de **system_call** es el número de la llamada al sistema
 * a invocar; se indicará en el registro EAX.
 * System_call permite hasta cuatro argumentos mas que serán pasados a
 * lo largo de la llamada al sistema. Uno de los argumentos puede ser
 * un puntero a una estructura */

244 ENTRY(system_call)
245 pushl %eax # save orig_eax

/* SAVE_ALL hace una copia de todos los registros de la CPU en la pila.
* Cualquier función de C que sea llamada encontrará sus argumentos en
- la pila porque SAVE_ALL los pone allí */

246 SAVE_ALL

System_call

```
/* Comprueba si la llamada al sistema esta haciendo una traza. Algunos
 * programas permiten hacer trazas en las llamadas al sistema como una
 * herramienta para curiosos o para obtener información extra de depuración */
247         GET_THREAD_INFO(%ebp)
248                                 # system call tracing in operation
249         testb
      $(_TIF_SYSCALL_TRACE|_TIF_SYSCALL_AUDIT),TI_flags(%ebp)
250jnz syscall_trace_entry                    ■

/* Compara si el numero de la llamada al sistema (EAX) excede del numero
 * máximo de llamadas al sistema */
251         cmpl $(nr_syscalls), %eax
252         jae syscall_badsys
```

System_call

/* Llama a la función del sistema.

Sys_call_table es una tabla de punteros a funciones del núcleo que implementan las llamadas al sistema. */

/* El segundo conjunto de paréntesis en la línea contiene tres argumentos:

* la dirección base del vector,
* el índice (EAX, el numero de la llamada al sistema),
* y la escala o numero de bytes en cada elemento del vector. */

```
253 syscall_call:
254         call *sys_call_table(,%eax,4)
```

System_call

/* La etiqueta **restore_all** es el punto de salida para **system_call**.

 * Su contenido es simple como la macro **RESTORE_ALL**, que

 * recupera los argumentos guardados por **SAVE_ALL** y retorna

 * al llamador del **system_call**. */

263 restore_all:

264 RESTORE_ALL

System_call

```
/* Comprueba si no existe trabajo pendiente */
276         movl TI_flags(%ebp), %ecx
277         andl $_TIF_WORK_MASK, %ecx  # is there any work to be done other
278                                     # than syscall tracing?
279         jz restore_all
/* Comprueba si se tiene que volver a asignar la CPU */
280         testb $_TIF_NEED_RESCHED, %cl
281         jnz work_resched
282
/* Comprueba si tiene señales pendientes y vuelve al modo núcleo modo virtual */
283 work_notifysig:                     # deal with pending signals and
284                                     # notify-resume requests
285         testl $VM_MASK, EFLAGS(%esp)
286         movl %esp, %eax
287         jne work_notifysig_v86      # returning to kernel-space or
288                                     # vm86-space
/* Comprueba si existe trabajo pendiente */
289         xorl %edx, %edx
290         call do_notify_resume
291         jmp restore_all
292
```

System_call

```
/* Si retornamos a modo virtual llamamos a save_v86_state */
294 work_notifysig_v86:
295         pushl %ecx              # save ti_flags for do_notify_resume
296         call save_v86_state       # %eax contains pt_regs pointer
297         popl %ecx
298         movl %eax, %esp
299         xorl %edx, %edx
300         call do_notify_resume
301         jmp restore_all
302
/* Punto de entrada si la llamada al sistema del proceso actual
 * esta haciendo una traza */
303         # perform syscall exit tracing
304         ALIGN
305 syscall_trace_entry:
306         movl $-ENOSYS,EAX(%esp)
307         movl %esp, %eax
308         xorl %edx,%edx
309         call do_syscall_trace
310         movl ORIG_EAX(%esp), %eax
311         cmpl $(nr_syscalls), %eax
312         jnae syscall_call
313         jmp syscall_exit
314
315         # perform syscall exit tracing
```

System_call

```
316         ALIGN
/* comprueba si hay trabajo pendiente si se vuelve del modo traza */
317 syscall_exit_work:
318         testb
    $(_TIF_SYSCALL_TRACE|_TIF_SYSCALL_AUDIT|_TIF_SINGLESTEP),
    %cl
319         jz work_pending
/* Habilita las interrupciones */
320         sti                 # could let do_syscall_trace() call
321                             # schedule() instead
322         movl %esp, %eax
323         movl $1, %edx
/* se llama al modo traza */
324         call do_syscall_trace
/* se vuelve al modo usuario */
325         jmp resume_userspace
326
327         ALIGN
```

System_call

```
/* si se produce un error guardar el estado, obtener la
    información
 * del hilo y volver al espacio de usuario */
328 syscall_fault:
329         pushl %eax                      # save orig_eax
330         SAVE_ALL
331         GET_THREAD_INFO(%ebp)
332         movl $-EFAULT,EAX(%esp)
333         jmp resume_userspace
334
335         ALIGN
/* si la llamada no es correcta volver al espacio de usuario */
336 syscall_badsys:
337         movl $-ENOSYS,EAX(%esp)
338         jmp resume_userspace
339
```

posiciones de los registros en en el stack

Después de volver de una llamada, muchos procedimientos necesitan esta información en el stack.

El registro EBX contiene la variable “current” la dirección que apunta a la entrada en la tabla de procesos del proceso que realiza la llamada.

```
51 EBX                  = 0x00
52 ECX                  = 0x04
53 EDX                  = 0x08
54 ESI                  = 0x0C
55 EDI                  = 0x10
56 EBP                  = 0x14
57 EAX                  = 0x18
58 DS                   = 0x1C
59 ES                   = 0x20
60 ORIG_EAX             = 0x24
61 EIP                  = 0x28
62 CS                   = 0x2C
63 EFLAGS               = 0x30
64 OLDESP               = 0x34
65 OLDSS                = 0x38
```

```
66
67 CF_MASK              = 0x00000001
68 IF_MASK              = 0x00000200
69 NT_MASK              = 0x00004000
70 VM_MASK              = 0x00020000
71
72 /*
73  * these are offsets into the task-struct.
74  */
75 state                =  0
76 flags                =  4
77 sigpending           =  8
78 addr_limit           = 12
79 exec_domain          = 16
80 need_resched         = 20
```

SAVE_ALL

```
#define SAVE_ALL
    cld;
    pushl %es;
    pushl %ds;
    pushl %eax;
    pushl %ebp;
    pushl %edi;
    pushl %esi;
    pushl %edx;
    pushl %ecx;
    pushl %ebx;
    movl $(__USER_DS),%edx;
    movl %dx,%ds;
    movl %dx,%es;
```

RESTORE_ALL
(RECUPERA EL ESTADO DEL PROCESO INTERRUMPIDO)

```
99 #define RESTORE_INT_REGS \
100         popl %ebx;      \
101         popl %ecx;      \
102         popl %edx;      \
103         popl %esi;      \
104         popl %edi;      \
105         popl %ebp;      \
106         popl %eax
107
108 #define RESTORE_REGS    \
109         RESTORE_INT_REGS; \
110 1:      popl %ds;       \
111 2:      popl %es;       \
112 .section .fixup,"ax";   \
113 3:      movl $0,(%esp); \
114         jmp 1b;         \
115 4:      movl $0,(%esp); \
116         jmp 2b;         \
117 .previous;              \
```

RESTORE_ALL

```
125 #define RESTORE_ALL
126         RESTORE_REGS
127         addl $4, %esp;
128 1:      iret;
129 .section .fixup,"ax";
130 2:      sti;
131         movl $(__USER_DS), %edx;
132         movl %edx, %ds;
133         movl %edx, %es;
134         movl $11,%eax;
135         call do_exit;
136 .previous;
137 .section __ex_table,"a";
138         .align 4;
139         .long 1b,2b;
140 .previous
141
```

SYS_CALL_TABLE 1

- Tabla con todas la funciones que tratan las llamadas al sistema.

```
578 ENTRY(sys_call_table)
579         .long sys_restart_syscall       /* 0 - old "setup()" system call,
    used for restarting */
580         .long sys_exit
581         .long sys_fork
582         .long sys_read
583         .long sys_write
584         .long sys_open          /* 5 */
585         .long sys_close
586         .long sys_waitpid
587         .long sys_creat
588         .long sys_link
589         .long sys_unlink        /* 10 */
590         .long sys_execve
```

SYS_CALL_TABLE 2

```
591         .long sys_chdir
592         .long sys_time
593         .long sys_mknod
594         .long sys_chmod         /* 15 */
595         .long sys_lchown16
596         .long sys_ni_syscall    /* old break syscall holder */
597         .long sys_stat
598         .long sys_lseek
599         .long sys_getpid        /* 20 */
600         .long sys_mount
601         .long sys_oldumount
602         .long sys_setuid16
603         .long sys_getuid16
604         .long sys_stime         /* 25 */
605         .long sys_ptrace
606 .long sys_alarm
```

…

SYS_CALL_TABLE 3

```
853         .long sys_mbind
854         .long sys_get_mempolicy
855         .long sys_set_mempolicy
856         .long sys_mq_open
857         .long sys_mq_unlink
858         .long sys_mq_timedsend
859         .long sys_mq_timedreceive       /* 280 */
860         .long sys_mq_notify
861         .long sys_mq_getsetattr
862         .long sys_ni_syscall            /* reserved for kexec */
863         .long sys_waitid
864         .long sys_ni_syscall            /* 285 */ /* available */
865         .long sys_add_key
866         .long sys_request_key
867         .long sys_keyctl
868
869 syscall_table_size=(.-sys_call_table)
```

Excepciones (1)

Son interrupciones producidas por la propia CPU cuando se producen ciertas situaciones como división por cero, desbordamiento del stack, fallo de página etc.

Como todas las interrupciones el núcleo realiza los siguientes pasos:

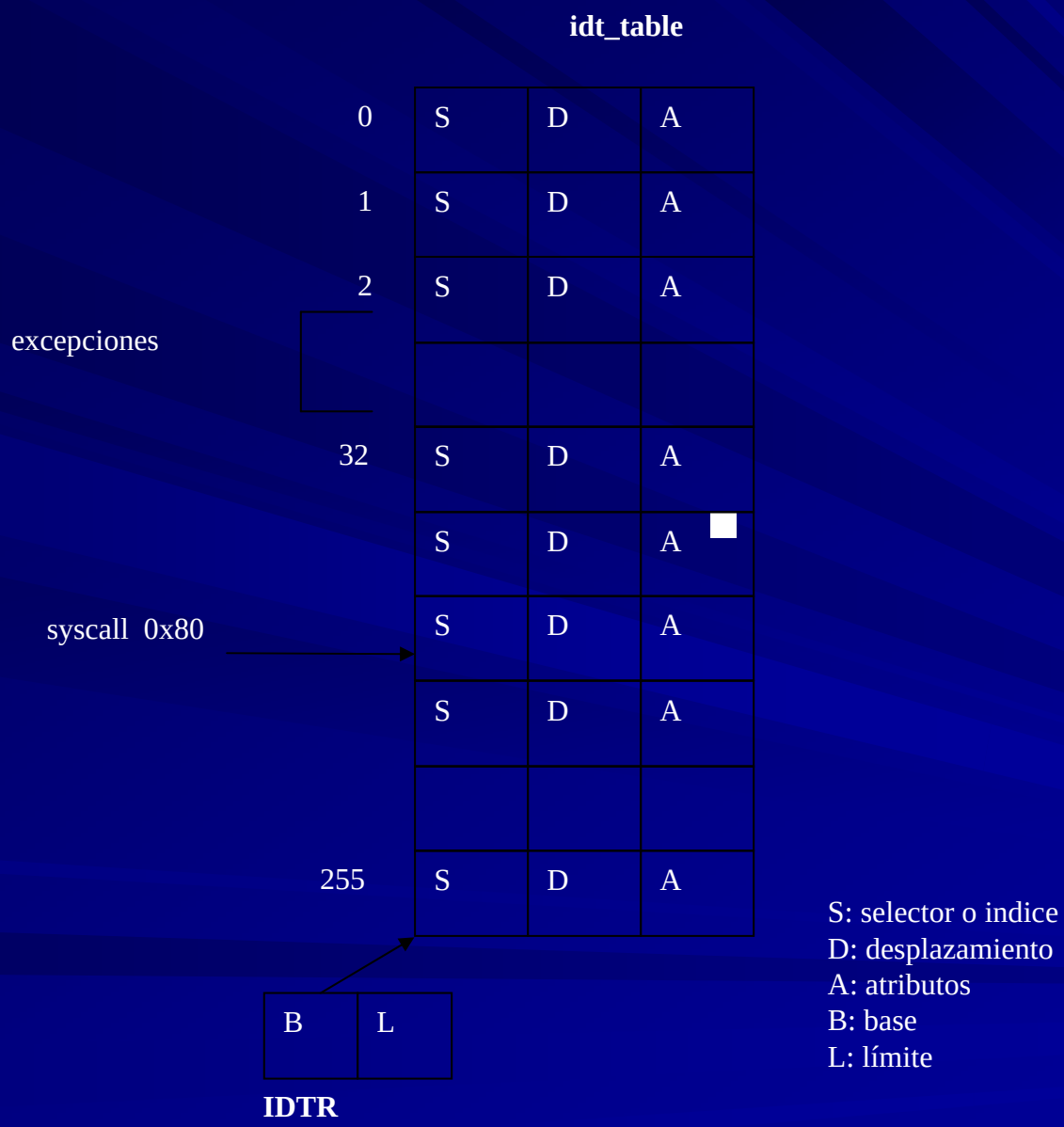
1. Guardar el estado del proceso en el stack.
   pushl, pushl,…
2. Llamar la función manejadora de la excepción.
   call …
3. Recuperar el estado de la función.
   pop, pop,…

Excepciones (2)

- Cuando se produce una excepción, la propia CPU realiza una instrucción de interrupción INT 0xNN para saltar dentro de la tabla IDT (tabla de descriptores de segmento de interrupciones) en la posición NN.

- En esta posición se encuentra un selector a la GDT (tabla global de descriptores de segmento) y un desplazamiento para alcanzar la dirección dentro del núcleo de la función (punto de entrada en el núcleo) que hay que ejecutar cuando se produce esta excepción.

Excepciones (3)



Excepciones (4)

```
378 ENTRY(divide_error)
379         pushl $0                    # no error code
380         pushl $do_divide_error
381         ALIGN
382 error_code:
383         pushl %ds
384         pushl %eax
385         xorl %eax, %eax
386         pushl %ebp
387         pushl %edi
388         pushl %esi
389         pushl %edx
390         decl %eax                   # eax = -1
391         pushl %ecx
392         pushl %ebx
393         cld
394         movl %es, %ecx
```

Excepciones (5)

```
/* almacenar la dirección de la función que atiende la excepción en
     el registro edi */
395         movl ES(%esp), %edi            # get the function address
396         movl ORIG_EAX(%esp), %edx      # get the error code
397         movl %eax, ORIG_EAX(%esp)
398          movl %ecx, ES(%esp)

/* preparar registros de la CPU para realizar la llamada */
399         movl $(__USER_DS), %ecx
400         movl %ecx, %ds
401         movl %ecx, %es
402          movl %esp,%eax                 # pt_regs pointer

/* llamada a la función que atiende la excepción */
403         call *%edi
```

Excepciones (6)

```
/* recuperar el proceso interrumpido */
404         jmp ret_from_exception
405
```

Veamos como se recupera el estado del proceso interrumpido en una excepción: ■

```
159 ret_from_exception:
/* desabilita interrupciones preempt_stop se define como cli */
160         preempt_stop
161 ret_from_intr:
162         GET_THREAD_INFO(%ebp)
163         movl EFLAGS(%esp), %eax         # mix EFLAGS and CS
164         movb CS(%esp), %al
```

Excepciones (7)

```
/* si es necesario volver al modo núcleo */
165         testl $(VM_MASK | 3), %eax
166         jz resume_kernel                # returning to kernel or vm86-space

167 ENTRY(resume_userspace)
168         cli                             # make sure we don't miss an interrupt
169                                         # setting need_resched or sigpending
170                                         # between sampling and the iret
171         movl TI_flags(%ebp), %ecx
172         andl $_TIF_WORK_MASK, %ecx      # is there any work to be done on
173                                         # int/exception return?
174         jne work_pending

/* salta a recuperar el estado del proceso interrumpido */
175         jmp restore_all
```

Excepciones (8)

Veamos el resto de puntos de entrada en el núcleo de las excepciones.

```
406 ENTRY(coprocessor_error)
407         pushl $0
408         pushl $do_coprocessor_error
409         jmp error_code
410
411 ENTRY(simd_coprocessor_error)
412         pushl $0
413         pushl $do_simd_coprocessor_error
414         jmp error_code
```

Excepciones (9)

```
511 ENTRY(int3)
512         pushl $-1                   # mark this as an int
513         SAVE_ALL
514         xorl %edx,%edx             # zero error code
515         movl %esp,%eax              # pt_regs pointer
516         call do_int3
517         testl %eax,%eax
518         jnz restore_all
519         jmp ret_from_exception
520
521 ENTRY(overflow)
522         pushl $0
523         pushl $do_overflow
524         jmp error_code
525
526 ENTRY(bounds)
527         pushl $0
528         pushl $do_bounds
529         jmp error_code
530
531 ENTRY(invalid_op)
532         pushl $0
533         pushl $do_invalid_op
534         jmp error_code
```

Excepciones (10)

```
541 ENTRY(invalid_TSS)
542         pushl $do_invalid_TSS
543         jmp error_code
544
545 ENTRY(segment_not_present)
546         pushl $do_segment_not_present
547         jmp error_code
548
553 ENTRY(general_protection)
554         pushl $do_general_protection
555         jmp error_code
556
/* punto de entrada cuando se produce un fallo de página, esto es la CPU
    necesita datos y no están en memoria principal y necesita traerlos de
    memoria segundaria */
561 ENTRY(page_fault)
562         pushl $do_page_fault
563         jmp error_code
564
```