Introducción a procesos

Luego de tener el kernel corriendo se quieren agregar procesos.
Estos procesos son código de usuario que es llamado a través de syscalls. En este tp se ven mecanismos para hacer cambios de contextos para poder lanzar UN proceso.

Procesos en JOS

Procesos es un término genérico. En JOS un proceso es un enviroment.

```
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 environments
   unsigned env_status;         // Status of the environment
   uint32_t env_runs;       // Number of times environment has run

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

El trap frame es el estado del CPU en un proceso para facilitar el cambio de contexto. ¿Por qué se llama trap frame? Viene de la arquitectura x86.
Cada proceso tiene un Page Directory. Cada proceso tiene un espacio de direcciones virtuales.

En el array de environments se guardan environments. **Cada environment guarda la metadata de un proceso**. Cuando la metadata desaparece el proceso muere. Los envs en el array no necesariamente están corriendo, pero si están vivos y con un estado. Para este TP, cómo implementamos el lanzamiento de UN proceso, esto significa que a lo sumo un proceso estará en ejecución, los demás estarán vivos pero no ejecutados.

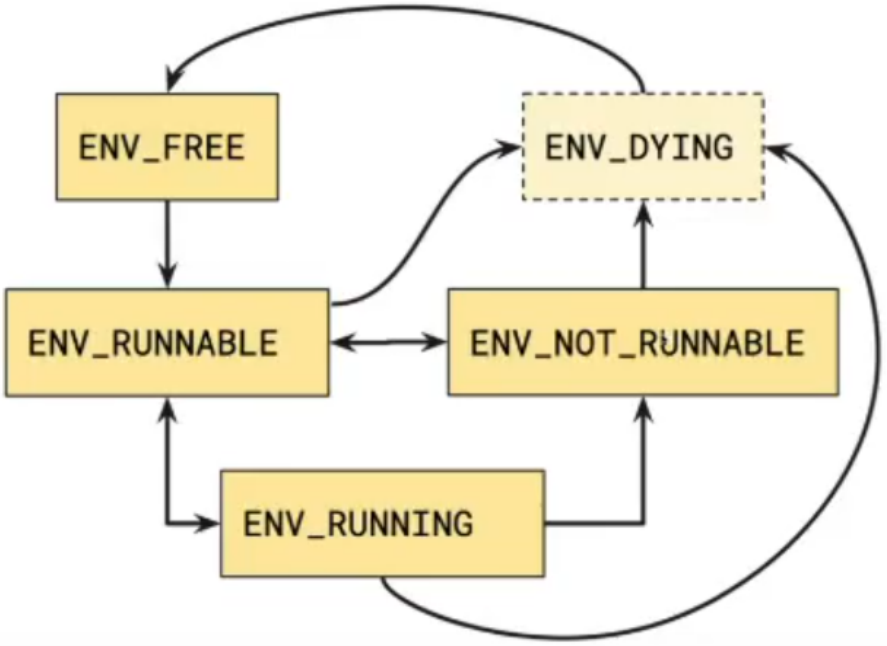
```
struct Env *envs = NULL;           // All environments
struct Env *curenv = NULL;         // The current env
static struct Env *env_free_list;  // Free environment list
                                   // (linked by Env->env_link)
```

Tiene la estructura de lista enlazada para ubicar los procesos, **curenv** es el proceso que está actualmente en la CPU corriendo.

Procesos Zombie

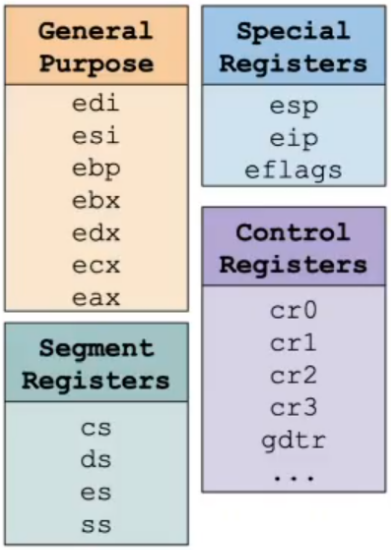
Un **proceso zombi** es un proceso en donde la metadata queda aunque ya termino. (en JOS no pasa esto, porque no se implementa la syscall wait())

Estados de un proceso



FREE: Un env libre, listo para cargar metadata.
RUNNABLE: No corriendo pero puedo correr.
RUNNING: Corriendo en el CPU (proceso activo en el CPU).
NOT_RUNNABLE: No corriendo y tampoco podría estar corriendo porque estoy bloqueado (Ej: I/O, gdb).
ENV_DYING: En múltiples CPUs, si yo mato a un proceso que estaba corriendo en otra CPU, se lo conoce como dying.

Trapframe y arquitectura x86



- **General Purpose:** Los podes usar para lo que quieras (los de toda la vida)
- **Special Registers:**
  - esp = StackPointer apunta a donde esta el stack
  - eip = InstructionPointer apunta a la instrucción que vamos a ejecutar en todo momento
  - eflags = flags de status, estos flags se usan para por ejemplo detectar overflow, división por cero, y cosas muy particulares.
- **Segment Registers**: Se utilizan para segmentación para proteger acceso a memoria en modo protegido. Te dice en qué dirección de memoria empieza el segmento al que corresponden a:
  - cs = CodeSegment instrucciones de código
  - ds = DataSegment datos para registros
  - es = ExtendedSegment instrucciones en particular (no importante)
  - ss = StackSegment stack

Nivel de privilegios

Se utiliza mucho para ver permisos (muy importante). El nivel de privilegios se almacena en los últimos 3 bits del cs (CodeSegment) llamados **CPL (CurrentPrivelegeLevel)**. 00 ring 0, 11 ring 3. Para cambiar de ring 00 a ring 11 se cambian estos bits.

Registros

- **Control Registers:**
  - cr0 tiene flags de activación por ejemplo paginación, páginas grandes, segmentación.

- cr3 almacena la dirección física donde está la tabla de paginación que va a usar la MMU para traducir.
- gdtr: global descriptor table, memoria física donde va estar la metadata de los segmentos y su respectiva dirección de memoria que luego va ser almacenada por ejemplo, en el cs (CodeSegment).

IMPORTANTE: El trap frame guarda todos los registros menos los de control.
Salvo el cr3 que se guarda en :

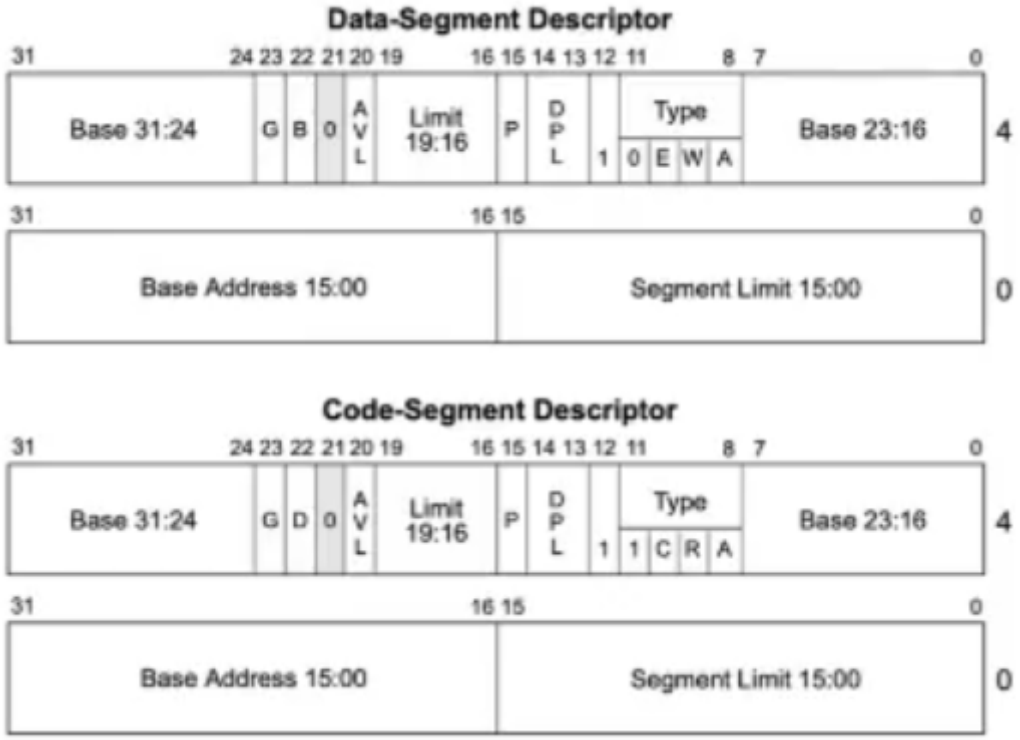
```
pde_t *env_pgdir;         // Kernel virtual address of page dir
```

Modificar el registro CS es una instrucción privilegiada y no se puede cambiar como por ej el instruction pointer eip.

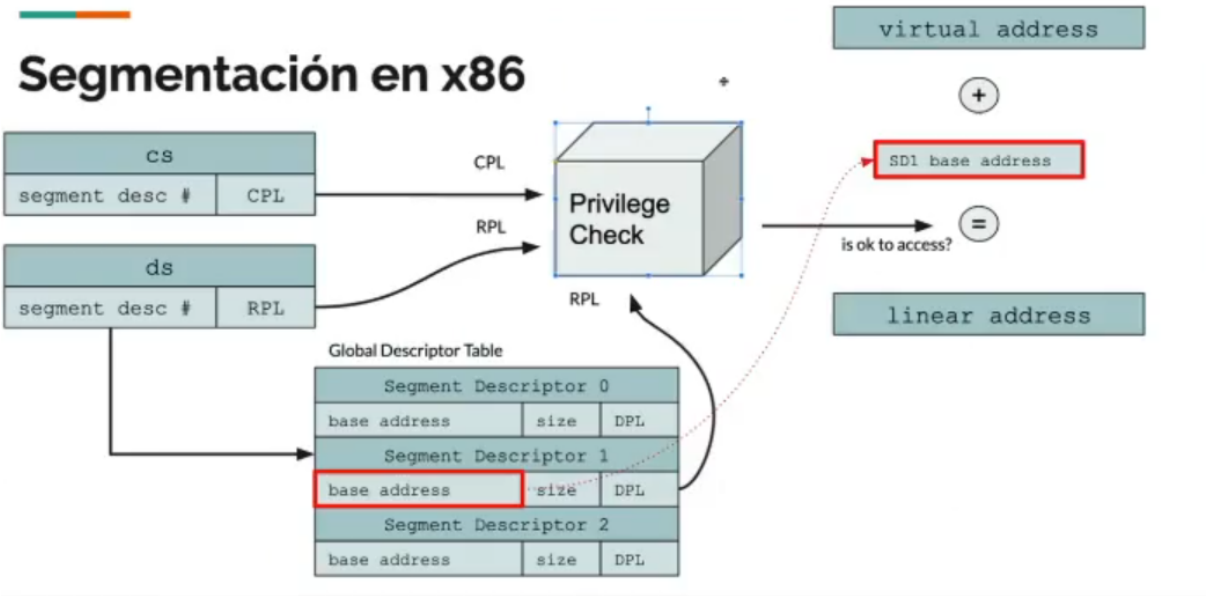
Segmentación en x86

En modo protegido, se tiene segmentación activada.
Un segmento es una región de la memoria. Los segmentos están definidos en una dirección que está guardada en **gdtr** (control register) y también los descriptores van a tener un **DPL (Descriptor Privilege Level)** que indica el permiso necesario para entrar.

Estructura descriptor:



Los segmentos están definidos tipo Base&Bound, el DPL son los permisos que requiere el acceso al segmento.



Si se quiere acceder a una dirección virtual para cargar un dato:

- Se usa el data segment
- Se toma el número de descriptor de segmento
- Se va a la tabla de descriptores globales.
- Busca el segmento y agarra la dirección base, el tamaño y el DPL
- Dirección base + dirección virtual = dirección lineal
- DPL se compara contra el CPL para ver tu privilegio actual en CS.
  Si falla “Segmentation Fault”

IMPORTANTE: Es fundamental para el chequeo de privilegios. Hoy en día no se usa Segmentacion para traduccion de direcciones, solo se usa la paginación para eso.
La segmentación se utiliza con base 0x00 y el size es 4GB. Abarca toda la memoria y no se hace traducción *(a casa pete).*

Por eso → V.A = L.A → V.A + 0x00 = L.A

GD_UT y GD_UD → CS y DS en user mode
GD_KT y GD_KD → CS y DS en kernel mode

Crear Environments

mem_init_envs()

Reserva memoria para el arreglo envs con boot_alloc() al igual que con UPAGES, pero en este caso UENVS.

env_init()

Similar mem_init() inicializando el arreglo envs. Itera NENV y setea todos los environments cómo libres, con un id en 0 y el env_link asociado al siguiente proceso.
Asigna un puntero a env_free_list del primer environment creado.

env_alloc()

Asigna e inicializa un nuevo entorno.
Busca un proceso libre de la lista de procesos libres, le configura un nuevo page directory y lo pone RUNNABLE. y los registros en 0. También asigna un nuevo ID de proceso (env_id) considerando incrementar el id anterior al primer proceso libre encontrado.

Setea los registros de segmento:

1. RPL: en 11 (ring 3, porque será un proceso de usuario) en todos los registros de segmento.

```
   e->env_tf.tf_ds = GD_UD | 3;
   e->env_tf.tf_es = GD_UD | 3;
   e->env_tf.tf_ss = GD_UD | 3;
   e->env_tf.tf_esp = USTACKTOP;
   e->env_tf.tf_cs = GD_UT | 3;
// Configurará e->env_tf.tf_eip más tarde.
```

2. También setea el stack pointer en una dirección virtual → USTACKTOP

Finalmente se quita de la lista de libres devolviendo el nuevo proceso alocado.

env_setup_vm()

Configura el page directory de un proceso nuevo. Este necesita tener mapeado todo el kernel, pages y envs.

IMPORTANTE: No puede ser el mismo page directory que el kernel, aunque este tiene todo lo que el nuevo proceso necesita mapeado desde el principio. El problema es que si le queremos agregar más cosas mapeadas, arruinamos el pgdir del kernel y este pgdir se acoplara con cada env. Por lo tanto necesito uno nuevo que sea una **COPIA** del kernpgdir. No se modifica el cr3 porque con las funciones sobre páginas se puede modificar este pgdir, porque queremos seguir usando, por ahora, el pgdir del kernel como el actual.

env_create()

Debemos implementar env_create() que esta wrappeada por una macro para poder usar los programas en el kernel. Es parecido a bootmain (este recorre el elf de la manera que nosotros necesitamos). Recibe un binario, se interpreta como un elf y se carga todo como se dice arriba.
El parseo del elf viene resuelto en el struct elf al que se castea el binario.
Esta llama a load_icode para cargar los program segments. (acá se hace el casteo)

region_alloc()

Garantiza que el program header va a entrar en una porción de memoria no alineada, pero que será tan grande como se solicite para poder alocar un program header.

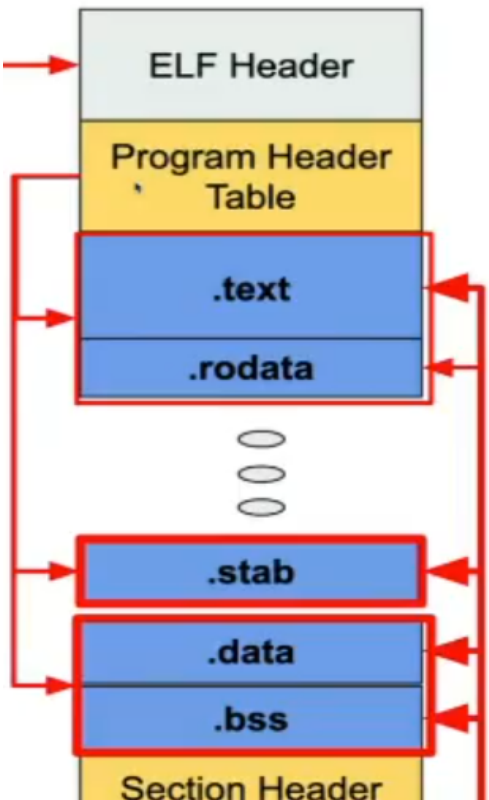
load_icode()

El load_icode no carga de disco como si carga el bootloader, justamente el readseg lee un pedazo de disco y eso se usa en el bootloader.

Un ELF es un arreglo de program headers los cuales deben ser cargados o no, dependiendo de un flag, en memoria.

Hay que tener cuidado con memcpy() y mem_set(), para ello debemos cambiar el cr3 al del proceso en donde queremos mapear la memoria. Si no lo cambias estarias memcopiandolo (? en el pgdir del kernel y ahi no tenemos la memoria allocada. Esto se hace con **lcr3(PADDR(e->env_pgdir)).**
Luego tenemos que volver al cr3 del kernel con lcr3(PADDR(kern_pgdir))

1. Se crea el ELF con el binary chequeando con la condición de e_magic = ELF_MAGIC
2. Se setea el program header (ph) en el direc: elf+elf→e_phoff
3. Se setea el final del program header en ph+elf→e_e_phnum
4. Cambiamos a pgdir del usuario (ring 3)
5. Por cada segmento del ELF se mapea una región con region_alloc (ph→p_va, ph→p_memsz).
    a. memcpy((void *) ph->p_va, binary + ph->p_offset, ph->p_filesz);
    b. memset((void *) ph->p_va + ph->p_filesz, 0, ph->p_memsz - ph->p_filesz);
6. Se mapea el ip del env a elf→e_entry
7. Como en el ELF no viene el stack, hay que mapearlo.
8. Se vuelve al pgdir del usuario (ring 0)



Formato ELF

ReadElf te da todos los datos, el endianness, arquitectura, los headers:

Program headers: cómo cargar el binario en memoria. Da la dirección en la que debes cargar un segmento y dice cuánto leer y desde donde en el binario elf.
(Se mapea desde offset hasta offset+size. La diferencia entre memsize y filesize es para dejar memoria iniciada a 0)

Section headers: indica cómo el binario puede ser enlazado. Especifica la dirección virtual de donde cargar cada programa.

IMPORTANTE: Como no tenemos filesystem, los programas vienen embebidos en la imagen del kernel.

Proceso de usuario

Este binario debe estar mapeado en el address space del proceso que se va a correr. Pero nosotros estamos en el address space del Kernel.

¿Qué quiere decir que sea el único address space? Significa que es el único libro de traducciones que tiene la MMU para traducir.

Pero nosotros queremos que el address space sea accesible desde un proceso de usuario. Por cada programa de usuario se creará un proceso nuevo que sale de la lista envs, y cada uno tendrá un address space propio.

El contenido deberá ser mapeado en el address space del usuario, ya que será quien lo necesite cuando esté corriendo.

¿Por qué sigue funcionando la vuelta al kernel luego de mapear la memoria del proceso? Porque el código original del kernel sigue estando presente en todos los procesos de usuario que se ejecuten. Todos los procesos tienen mapeado en su parte superior, los mapeos del kernel y por eso es una decisión de diseño tener el kernel duplicado.

Lanzar procesos

Debemos agarrar los registros y cargarlos todos en el CPU. Esto cambia el modo a ring 3. Para poder permitirle un llamado a syscall necesitamos una **interrupción**.
Se utilizan las siguientes funciones:

- env_run()
  - Marca el env cómo RUNNING
  - Actualiza el cr3
  - Llama el env_pop_tf
- env_pop_tf()
  - Hace el cambio de contexto y ya estás en modo usuario.
  - Agarra un struct tf que recibe por parámetro en el tope del stack.
  - Setea los registros uso general
  - Mueve el stack pointer al puntero que diga el tf.
  - Llama iret.

- iret()
    - Es una instrucción de assembler que es PRIVILEGIADA. Restaura eip, eflags, cs y esp.
    - Con eip salta a una sección de código a ejecutar.
    - Con esp cambia el stack de llamadas.
    - Con cs cambia el CPL, de ring 0 a ring 3 (inverso no, porque es privilegiada y no se puede hacer)
    - Básicamente se recupera la info del estado del proceso cuando se generó una interrupción antes

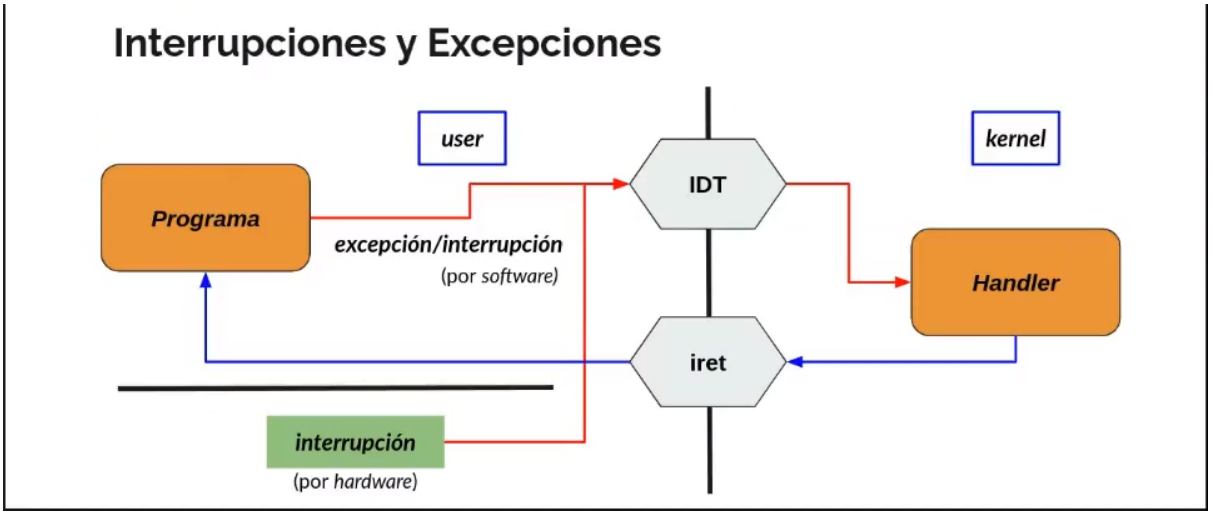
Interrupciones y syscalls

En esta parte del TP se desarrolla la vuelta de ring 3 a ring 0, es decir, cuando termina el proceso y se vuelve a modo kernel.

La primera pregunta es: cómo hacemos para que el proceso se comunique con el exterior?

Definiciones:

1. **Interrupciones**: son generadas debido a señales externas de dispositivos de hardware o ejecutadas por software. Algunas serán recuperables, otras no.
2. **Excepciones**: Ocurren cuando el CPU detecta un error en la ejecución de una instrucción.



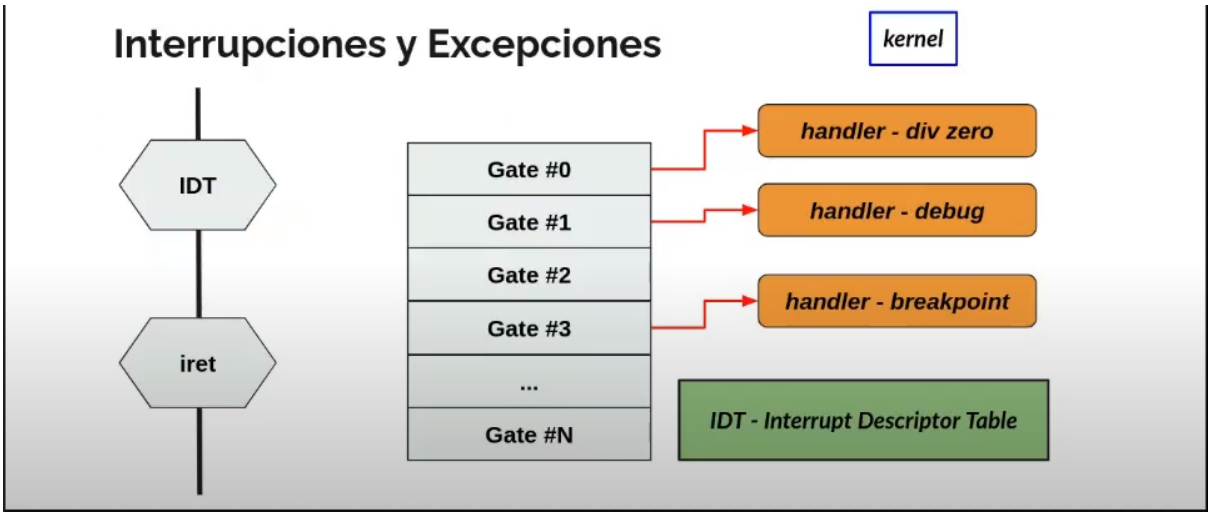
Siempre que llega una excepción o interrupción, para el kernel, son lo mismo y son diferentes a la vez. Es lo mismo porque siempre actuará de la misma manera, aunque sabe que provienen de diferentes canales.

Procedimiento:

1. Uno tiene un programa de usuario en **user-land** que genera una excepción o una interrupción por software, también puede ser por hardware. Pero todas van hacia lo que es una IDT (interrupt descriptor table)

2. Eso llega al kernel mediante un handler que maneja las diferentes interrupciones. Habrá uno por interrupción
3. Cuando el kernel termina de resolver, retorna al proceso que la ocasionó, mediante iret que vuelve al estado original como se vio en este tp.

Zoom parte derecha de la imagen superior



Las filas de esta tabla son como compuertas, por donde el proceso se mete y solicita hacer un syscall. Cada una de estas entradas tienen un handler diferente configurado para cada una de estas interrupciones.

La IDT es configurada por el kernel, según la convención de x86.
The Interrupt Descriptor Table (IDT): El procesador se asegura de que las interrupciones y las excepciones solo pueden hacer que se ingrese al kernel en algunos puntos de entrada específicos y bien definidos determinados por el propio kernel, y no por el código que se ejecuta cuando se toma la interrupción o la excepción.

El x86 permite hasta 256 puntos de entrada de interrupción o excepción diferentes en el núcleo, cada uno con un vector de interrupción diferente. Un vector es un número entre 0 y 255. El vector de una interrupción está determinado por la fuente de la interrupción: diferentes dispositivos, condiciones de error y solicitudes de aplicaciones al kernel generan interrupciones con diferentes vectores. La CPU usa el vector como un índice en la tabla de descriptores de interrupción (IDT) del procesador, que el kernel configura en la memoria privada del kernel, al igual que la GDT. Desde la entrada apropiada en esta tabla, el procesador carga:

- el valor a cargar en el registro del puntero de instrucción (EIP), apuntando al código del kernel designado para manejar ese tipo de excepción.
- El valor para cargar en el registro del segmento de código (CS), que incluye en los bits 0-1 el nivel de privilegio en el que se ejecutará el controlador de excepciones. (En JOS, todas las excepciones se manejan en modo kernel, nivel de privilegio 0).

Hay ciertas excepciones que tienen un nombre definido en x86, tiene un largo de 26 (creo) y después se podrían desarrollar interrupciones propias a nivel kernel.

1. Segment Not Present
2. Breakpoint
3. Overflow
4. División Error
5. Page Fault

En la tabla original se muestran de dónde viene e información importante.

trap_init()

trap_init será usada para manejar las interrupciones. Se deberá configurar la IDT, donde se definen cada uno de los handler para las interrupciones que manejaremos.

IDT está representada en JOS como **struct Gatedesc** en el **trap.c**

extern uint32_t trap_divzero;
Usar macro SETGATE(idt[T_DIVIDE], , GD_KT, trap_divzero, [0, 1, 2, 3]?);

segundo parámetro: no deja pasar ninguna interrupción hasta que se maneja la actual. Se maneja de forma secuencial y no pueden anidarse. Es una limitación de JOS.
dpl: nivel de acceso requerido para ejecutar por software esta interrupción (ult parámetro)

trap_init_percpu()

Contiene la clave de cómo funciona el ida y vuelta entre el user space y kernel land, particularmente lo siguiente:

ts.ts_esp0 = KSTACKTOP;
ts.ts_ss0 = GD_KD;

Interrupt Call Flow

Luego de ejecutar una interrupción el código llega a la CPU. La CPU procederá y le pasará el control al kernel. Este procedimiento consta en dejar información de contexto para que el kernel pueda manejar bien la interrupción.

El kernel hace un cambio de stack y aunque el programa termine, el kernel no destruye el stack viejo del proceso que falló. Este cambio de stack se realiza usando el ts.ts_esp0 definido en trap_init_percpu, que guardó el stack del kernel. Todo el código que se ejecutará será el del kernel. El GD_KD es la …

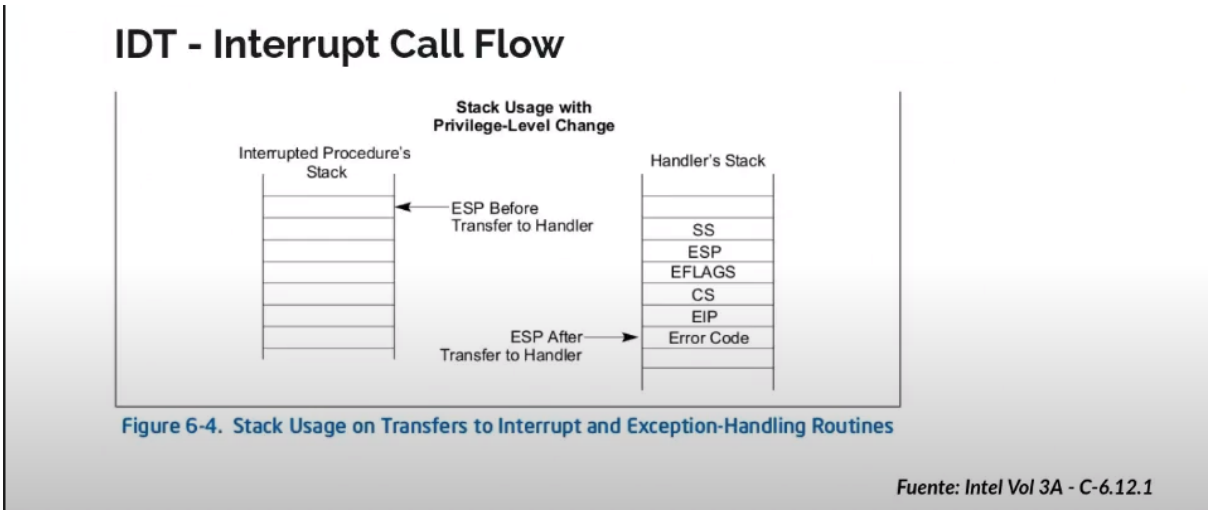
1. Vector interrupt significa que llega una interrupción que indexa la tabla IDT, se elige un posible handler.
2. Este handler tiene asociado un segmento donde tiene que correr.
3. Se busca el segmento y se verifica que esté en la GDT.
4. Se obtiene la base address + un index de la primera tabla.
5. Se saltará hacia el código de uno de estos handlers.

Las interrupciones suelen tener un nivel de privilegio más alto que cualquier otra tarea que tiene que resolver el CPU. Para resolver primero se guarda el contexto de lo que estaba haciendo. Entonces para no perder por ejemplo, el Instruction Pointer, la CPU utiliza el stack guardado en tss para pushear en un cierto orden ya definido, todos los registros importantes y alguna otra información (lo supongo yo).

Otras cosas que guarda:

1. SS
2. ESP
3. EFLAGS
4. CS: importante porque se puede saber en qué ring se estaba ejecutando código de usuario, por ejemplo.
5. EIP

Cuando hay un código de error, se hace un push adicional. Para estos errores hay determinadas interrupciones en la tabla ya predeterminada.



En JOS, el TrapFrame contiene tf_esp y tf_ss, tiene un padding de bits en 0 para que todos ocupen 32 para cuando el proceso está siendo ejecutando en un Ring de usuario. Además se lee de abajo hacia arriba, tiene un orden determinado intencionalmente.

Los que están por arriba de tf_esp son los registros que se pushean en el proceso del kernel.

```
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));
```

Macros:

Se debe definir cada interrupt handler en trapentry.S. Para no repetir demasiado código, se proporcionan las macros TRAPHANDLER y TRAPHANDLER_NOEC

TRAPHANDLER: pushea el número de interrupción, que está representado por el tf_trapno.

TRAPHANDLER_NOEC: es un poco diferente, pushea el 0 y luego el número de interrupción. Esto es para que tengan siempre el mismo formato, las interrupciones que no tienen error code son distintas entonces el error code es pasado como 0 para que lo cumpla.

Debajo del .text se deberían definir todos los handlers, ejemplo:

- TRAPHANDLER_NOEC(trap_divzero, T_DIVIDE)

kern_idt()

En JOS, todas las excepciones, interrupciones y traps se derivan a la función trap(), definida en trap.c. **Esta función recibe un puntero a un struct Trapframe como parámetro, por**

lo que cada interrupt handler debe, en cooperación con la CPU, dejar uno en el stack antes de llamar a trap().

_alltraps

Hay que pushear todos los valores que aún no han sido cargados… **pushl de %ds, %es**, y registros de propósito general (**pushal**, hace un push masivo de todos). También, hay que agregar en el medio algún padding si fuera necesario. Hay que agregar valores adecuados, esto significa que esto está siendo corrido del lado del kernel. Se carga GD_KD en %ds y %es con un movw después de pushearlos, para no sobreescribir el registro.

Para terminar de manejar las interrupciones se puede hacer una llamada desde Assembler. Por ejemplo:
pushl %esp (en C los argumentos están en el tope del stack y necesitamos un puntero al struct Trapframe, y este es el propio stack pointer)
call trap

Manejo de interrupciones

Handling Page Faults

The page fault exception, interrupt vector 14 (**T_PGFLT**), es particularmente importante. Cuando el procesador toma una falla de página, almacena la dirección lineal (es decir, virtual porque JOS hace un mapeo 1-1) que causó la falla en un registro de control de procesador especial, CR2. En trap.c proporcionamos los inicios de una función especial, page_fault_handler(), para manejar las excepciones de errores de página.

The Breakpoint Exception

The breakpoint exception, interrupt vector 3 (**T_BRKPT**), se usa normalmente para permitir que los depuradores inserten puntos de interrupción en el código de un programa reemplazando temporalmente la instrucción de programa relevante con la instrucción especial de int 3 interrupción de software de 1 byte. En JOS, abusaremos ligeramente de esta excepción al convertirla en una llamada de pseudo sistema primitivo que cualquier entorno de usuario puede usar para invocar el monitor del kernel de JOS. Este uso es en realidad algo apropiado si pensamos en el monitor del kernel de JOS como un depurador primitivo.

Syscalls

https://pdos.csail.mit.edu/6.828/2017/labs/lab3/

Los procesos de usuario le piden al kernel que haga cosas por ellos invocando llamadas al sistema. Cuando el proceso del usuario invoca una llamada al sistema, el procesador ingresa al modo kernel, el procesador y el kernel cooperan para guardar el estado del

proceso del usuario, el kernel ejecuta el código apropiado para llevar a cabo la llamada al sistema y luego reanuda el proceso del usuario. Los detalles exactos de cómo el proceso de usuario capta la atención del núcleo y cómo especifica qué llamada desea ejecutar varían de un sistema a otro.

Hoy en día, la mayoría de sistemas operativos implementan sus syscalls en x86 mediante las instrucciones SYSCALL/SYSRET (64-bits) o SYSENTER/SYSEXIT (32-bits). Tradicionalmente, no obstante, siempre se implementaron mediante una interrupción por software de tipo trap.
En JOS, se elige la interrupción n.º 48 (0x30) como slot para **T_SYSCALL**. Deberá configurar el descriptor de interrupción para permitir que los procesos de usuario provoquen esa interrupción. Tenga en cuenta que la interrupción 0x30 no puede ser generada por hardware, por lo que no hay ambigüedad causada por permitir que el código de usuario la genere.

La aplicación pasará el número de llamada al sistema y los argumentos de la llamada al sistema en los registros. De esta manera, el kernel no necesitará hurgar en la pila o el flujo de instrucciones del entorno del usuario. **El número de llamada del sistema irá en %eax, y los argumentos (hasta cinco de ellos) irán en %edx, %ecx, %ebx, %edi y %esi**, respectivamente. **El núcleo devuelve el valor de retorno en %eax.** El código ensamblador para invocar una llamada al sistema ha sido escrito para usted, syscall() en lib/syscall.c

Protección de memoria

En la implementación actual de algunas syscalls ¡no se realiza suficiente validación! Por ejemplo, con el código actual es posible que cualquier proceso de usuario acceda (imprima) cualquier dato de la memoria del kernel mediante sys_cputs()

sys_cputs: genera un acceso invalido de memoria pero del lado del kernel, entonces habría que protegerlo dentro del kernel por más que no arroje un segmentation fault al user.

user_mem_check()

- Llamar a panic() en trap.c si un page fault ocurre en el ring 0 (para determinar si ocurrió una falla en el modo de usuario o en el modo kernel, verifique los bits bajos del archivo tf_cs)
- Implementar user_mem_check():
    - Se recorre página por página desde va hasta va+len si el env tiene los permisos necesarios (verificar flags) o si va < ULIM.
    - Si hay error, se setea user_mem_check_addr a la primera addres con error.
- Para cada syscall que lo necesite, invocar a user_mem_assert() para verificar las ubicaciones de memoria.
- user_mem_assert():
    - llama a user_mem_check para chequear que tenga permisos
    - Si no tiene permisos, se **destruye** el env.