Parte práctica

TP2 Parte 1

Vamos a agregarle a JOS la capacidad de crear procesos y hacer que el SO haga cosas útiles. En el TP1 trabajamos desde modo kernel. Vamos a implementar algunas syscalls para usar en modo usuario y para pasar de modo usuario a modo kernel. Va a haber un único proceso porque nos queremos focalizar en esto. En el TP3 vamos a tener n procesos y vamos a agregar un scheduler para que administre la ejecución de los procesos.

TP2 → procesos de usuario, único proceso, mecanismos para salir y entrar del kernel
Puede pasar que se vayan acumulando los errores desde el primer al último TP. Para minimizar estos errores tener en cuenta las precondiciones de las funciones.

Introducción a procesos
El address space de un proceso es un mapeo de memoria virtual a memoria física. Con lo cual al ver heap, stack, code tenemos que ver que el proceso va a tener su propio directorio de páginas, va a tener un page directory y dado que JOS usa paginación va a tener sus propias traducciones de direcciones virtuales y espacio de direcciones virtuales.

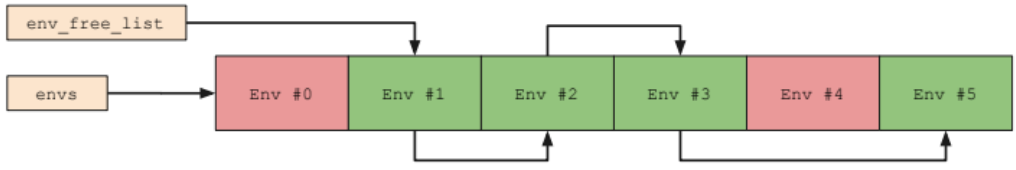
El concepto proceso es *JOS* se llama **environment**.

La estructura en donde se guardan los procesos es process table o pro cess control block. Es un TDA con elementos que representan un proceso, es decir, guarda toda la data necesaria para poder mantener, correr un proceso.
En JOS vamos a tener una estructura de dato struct **ENV** que tiene toda la data referenciada a un proceso.

```
struct Env {
  struct Trapframe env_tf;
  struct Env *env_link;
  envid_t env_id;
  envid_t env_parent_id;
  enum EnvType env_type;
  unsigned env_status;
  uint32_t env_runs;

  // Address space
  pde_t *env_pgdir;
};
```

Tiene un id, un parent id (todo proceso va a tener un padre), estado.
env_id →es el número del proceso (process id)
env_parent_id →el número del proceso padre
env_link → se usa como una forma de lista enlazada
env_type → para diferenciar distintos procesos entre sí. Por ahora vamos a tener un único tipo de proceso. Va a ser útil a futuro
env_status →ciclo de estado de un proceso
env_runs → es un contador que mantiene el kernel para saber cuántos time slices de CPU tuvo un proceso.
env_pgdir → cada proceso tiene su propio page directory. Esta es la gracia de la paginación. La paginación no sirve si siempre usamos el mismo esquema de traducciones. Es útil cuando podemos tener distintos espacios de direcciones virtuales. Un page directory codifica todo el espacio de direcciones virtuales, y tener varios page directory permite tener varios espacios de direcciones virtuales a las que mapear, ergo varios procesos
env_tf → es el estado del CPU en un proceso al momento de "congelarlo". Se necesita guardar el estado de los registros (el stack pointer y todos los demás registros).

Process Control Block
Tiene una pinta muy parecida al struct pages. Es un arreglo donde cada elemento representa un proceso y está indexado. **Envs** es un arreglo de 1024 environment. Además, hay una lista enlazada de environments libres, es decir, posiciones en el arreglo envs donde yo puedo guardar un proceso nuevo. Env_free_list apunta a el primer proceso libre dentro de envs



En este ejemplo, los procesos 1, 2, 3 y 5 no están corriendo por lo que no son procesos reales. Son espacios vacíos en donde el kernel puede guardar un proceso nuevo. Cuando uno quiere crear un proceso, el kernel tiene que averiguar un nuevo id e inicializar un struct env nuevo para ese nuevo proceso, lo tiene que guardar en algún lado y para esto debe encontrar una posición sin usar dentro de este arreglo y guardarlo en esa posición. La posición del arreglo envs no es lo mismo que el id del proceso. Los id de procesos se generan con otra lógica. La posición del arreglo no tiene nada que ver con el process id que va a terminar teniendo el proceso.
Para que se busque una posición en el arreglo de forma eficiente, se usa una lista enlazada pues facilita encontrar en tiempo constante una posición libre. Tenemos una cantidad

máxima de procesos que está dada por el tamaño del arreglo que viene definido en la constante NENVS que indica cuántos procesos puede albergar el arreglo envs. El arreglo envs tiene una cantidad fija de posiciones y es 1024 (2^10).

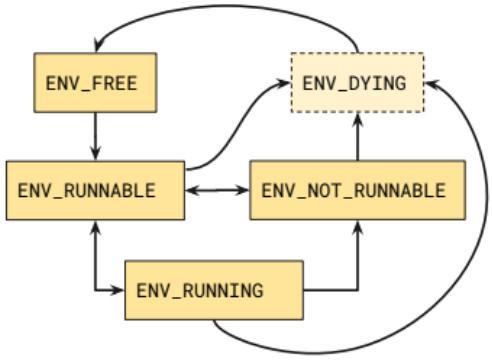
```
#define NENV ....
struct Env *envs;
struct Env *curenv
static struct Env *env_free_list;
```

El arreglo envs está definido como una variable global así como la variable que apunta a la cabecera de la lista.
struct Env *currenv → puntero al elemento del arreglo, es decir al proceso, que está siendo ejecutado en el CPU

Diagramas de estados en los procesos de JOS

Procesos en el JOS - Estados

- Los procesos en JOS tienen 5 estados posibles
  - **Free**: el struct Env está en la lista de procesos libres
  - **Runnable**: el proceso está listo para ser ejecutado en la CPU (hasta que el scheduler lo elija)
  - **Not runnable**: bloqueado esperando alguna operación. No elegible por el scheduler
  - **Running**: actualmente en el CPU
  - **Dying**: estado especial *opcional* si un proceso es terminado mientras estaba corriendo *en otro CPU*

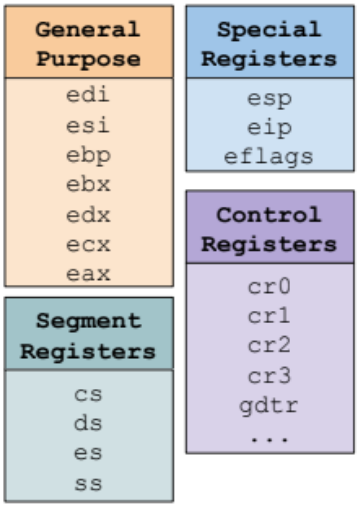


Los 3 estados centrales son: **runnable, running y not runnable.**
running → estoy corriendo en la CPU
not runnable → no puedo correr en la CPU porque estoy bloqueado por alguna operación
runnablo → estoy en condiciones de correr en la CPU pero ahora hay otro proceso corriendo

Los otros 2 estados tienen que ver con la operatoria de los procesos en JOS.
free → esta posición del arreglo env está libre para ser tomada por un proceso nuevo
dying → es una especie de proceso zombie que se usa si un proceso muere en un lado pero está viviendo en otro

Los registros en x86
Registro propósito general → se pueden usar para hacer cuentas, acceder a memoria y el programador tiene completo control sobre estos registros
Registro especiales → se pueden y no modificar
Registro de control → interactuamos con ellos en la parte de paginación. Tienen bits que controlan distintas partes de cómo funciona el CPU. Cuando tocamos el cr3 estamos modificando el lugar en que la MMU va a buscar el directorio de páginas. Cuando cambiamos un bit en cr0 estamos cambiando si usamos paginación, páginas grandes, etc. Pero hay más registros de control que hacen referencia a los registros de segmento. Indican

a la CPU dónde ir a buscar ciertas estructuras de datos que van a estar en memoria y con este registro controlamos cómo funciona la segmentación
Registro de segmentación → en x86 tenemos al menos 4: code, data, extra y stack segment. Estos registros controlan la segmentación.



Los segmentos en x86 no se van a usar para traducir direcciones virtuales a físicas pues esto se hace con paginación. La segmentación está para controlar permisos a nivel hardware (ring) específicamente del code segment (cs). En el code segment los últimos 2 bits indican en qué ring estamos. Ejemplo: si los últimos 2 bit son 00 estamos en el ring 0, si los bit son 11 estamos en el ring 3.
Entonces para saber en qué ring se está ejecutando el procesador de la arquitectura x86, vemos estos 2 últimos bits del cs. Solo vamos a poder modificar el registro cs bajo ciertas condiciones especiales y vamos a requerir privilegios. Vamos a ver qué instrucciones modifican este registro. Si queremos pasar de modo kernel a usuario o viceversa vamos a tener que cambiar esos 2 bits del cs.

El cs tiene contenido una dirección física de una sección de memoria, que el stack pointer va a buscar para encontrar las instrucciones a ejecutar

Segmentación en x86
Los descriptores de segmentos van a describir a los segmentos, es decir, van a mantener la data de dónde empiezan, dónde terminan y algunos flags. Entonces los segmentos de x86 no tienen base and bounds sino un número de segmento que se usa para indexar en una tabla de descriptores de segmentos donde está la data. A esta tabla se la conoce como Global Descriptor Table (GDT). Es una tabla que está en memoria pero como el CPU la tiene que usar por cada instrucción que resuelva para resolver segmentación, necesitamos decirle al CPU dónde encontrar en memoria esa tabla. Para eso están los registros que mencionamos antes, hay uno que se llama GDTR que es un registro del CPU que se puede setear con una instrucción privilegiada y contiene la dirección física en la que empieza la GDT.

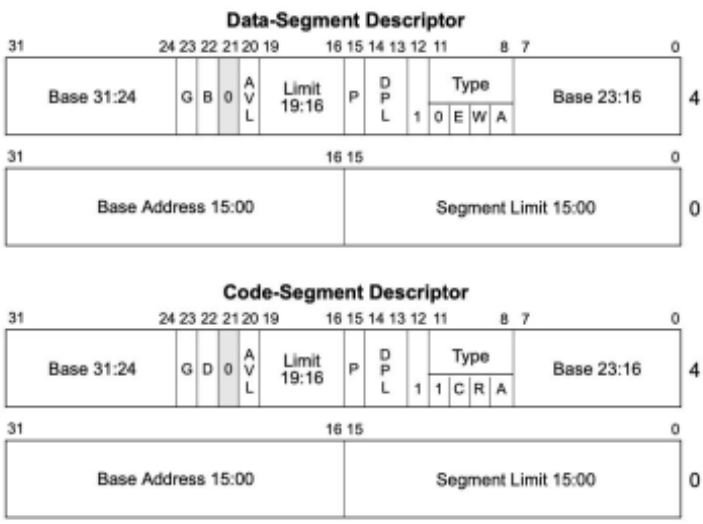
```
// Load GDT and segment descriptors.
void
env_init_percpu(void)
{
    lgdt(&gdt_pd);
    // The kernel never uses GS or FS, so we leave those set to
    // the user data segment.
    asm volatile("movw %%ax,%%gs" : : "a"(GD_UD | 3));
    asm volatile("movw %%ax,%%fs" : : "a"(GD_UD | 3));
    // The kernel does use ES, DS, and SS.  We'll change between
    // the kernel and user data segments as needed.
    asm volatile("movw %%ax,%%es" : : "a"(GD_KD));
    asm volatile("movw %%ax,%%ds" : : "a"(GD_KD));
    asm volatile("movw %%ax,%%ss" : : "a"(GD_KD));
    // Load the kernel text segment into CS.
    asm volatile("ljmp %0,$1f\n 1:\n" : : "i"(GD_KT));
    // For good measure, clear the local descriptor table (LDT),
    // since we don't use it.
    lldt(0);
}
```

Por lo que configurando esa tabla y los registros de segmentación podemos implementar la segmentación en x86.
Cada descriptor de segmentos contiene ciertos permisos. Estos se usan para indicar qué segmento está intencionado para ser accedido por el usuario y/o el kernel. Cuando el CPU quiera acceder a una dirección virtual primero pasa por la etapa de segmentación, va a comparar cuál es el privilegio actual, va a calcular el segmento de la dirección que estás intentando acceder, va a acceder al descriptor del segmento, se va a fijar cuáles son los permisos que tiene ese descriptor, compara los números. Si sos usuario y estás accediendo a un descriptor del kernel no va a funcionar, al revés funciona. Es como un chequeo de permisos, lo hace la MMU.

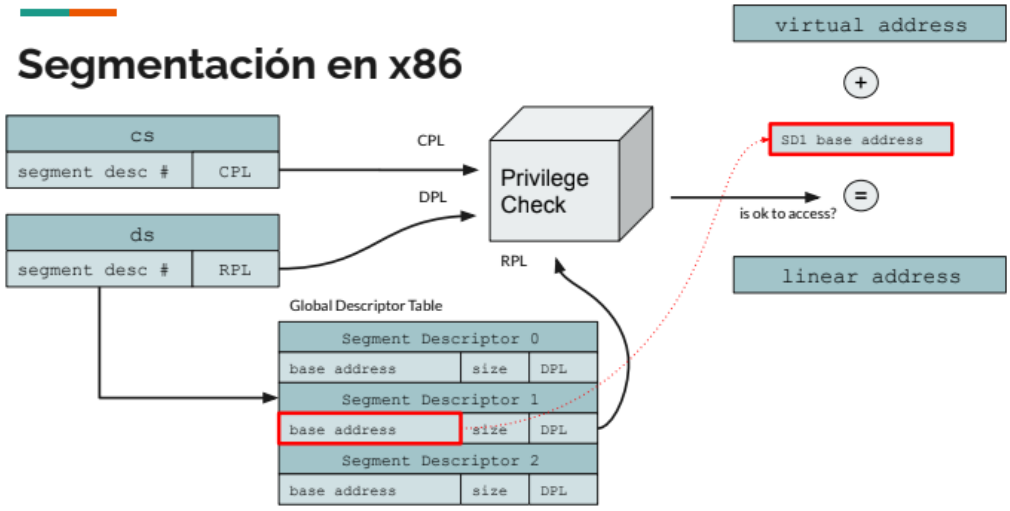
Esta es la pinta que tiene un descriptor de segmentos.

Ejemplo de descriptor

- La **base address** indica donde comienza el segmento en memoria física
- El **segment limit** indica el tamaño del segmento
- El **Descriptor Privilege Level** (DPL) indica qué permisos tiene que tener alguien (CPL) para acceder a ese segmento en particular



Esto es lo que ocurre cuando intentamos acceder.



Tenemos el code segment que define el nivel de privilegio actual, la tabla de descriptores y el segmento que se está usando (en este ejemplo es el data segment pero podría ser el code segment). Cada una de las instancias de la GDT tiene su propio bit de permisos. El módulo de segmentación hace la misma cuenta que hacíamos: base+offset, chequea que no esté por arriba del límite. Pero además se introduce un control de permisos que se fija si tu nivel de privilegio actual te deja acceder al segmento al cual estas intentando entrar. Si te deja pasa, sino levanta un seg fault.

En JOS todos los segmentos tienen base 0 y límite 4 GB con lo cual estamos mapeando a toda la memoria virtual en todos los segmentos. Esto quiere decir que la segmentación no va a cambiar al número de dirección a la cual estamos queriendo acceder. Eso lo va a hacer paginación.
Pero lo que tenemos en el medio es el chequeo de permisos.

trap_init() → interrupciones. Para volver de modo usuario a modo kernel
lgdt → carga GDT
SEG(arg1,arg2,arg3,arg4) → es una macro que recibe permisos, base, límite y privilegio en ese orden
Todos los segmentos tienen base 0 y límite 4 GB. Hay algunos que tienen permisos de ejecución y lectura, otros tienen permiso de escritura pero no de ejecución. Esto depende del segmento al cual hacemos referencia (code o data).

El registro TSS se va a usar más adelante y se encarga de hacer context switches.

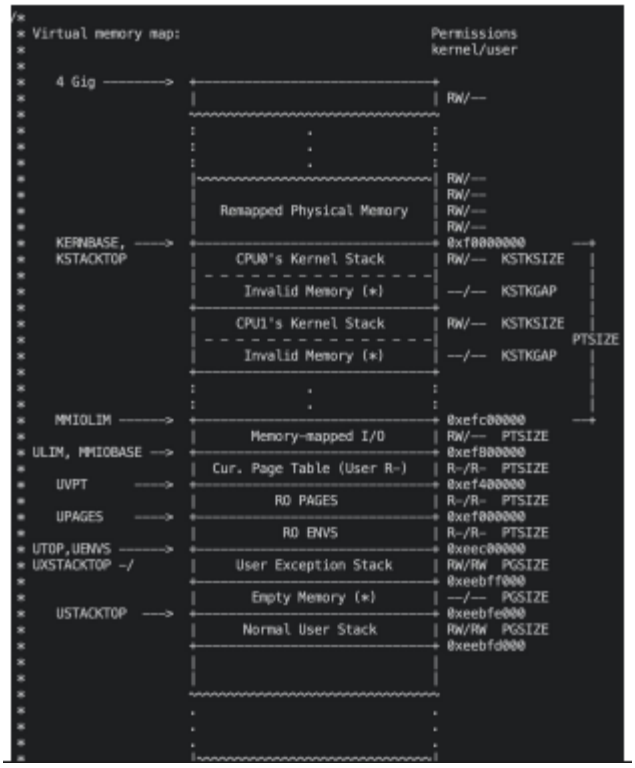
Ayuda: manual de intel.

Tarea 1 - El arreglo envs

El arreglo envs se crea con boot_alloc() al igual que pages. En arreglo env no va a llegar a ser 4 K sino que va a ser más chico.
En mem_init() se reserva memoria para env y en env_init() se inicializa el arreglo.
env_alloc() es el equivalente a page_alloc(). Agarra el arreglo envs, encuentra un elemento que esté marcado como libre y lo devuelve. Pero además hace un tipo de inicializaciones básicas y configura el id que va a tener el proceso. Esta función ya está implementada, hay que responder algunas preguntas acerca del código.

Estamos construyendo un espacio de direcciones en un proceso usuario. Todos los procesos usuarios de JOS van a tener el mismo formato en su espacio de direcciones virtuales.

El siguiente es el espacio de direcciones de todos los procesos de usuario.



Tiene regiones como UTEXT y USTACK donde se va a mapear el stack y el código del proceso. Además, hay más cosas mapeadas del estilo del kernel, stack del kernel, estructuras internas del kernel con permiso de solo lectura como pages y envs. El usuario siempre va a tener esto mapeado en su espacio de direcciones.
¿Por qué el kernel y su stack están mapeados en el espacio de direcciones de todos los procesos? El kernel va a estar mapeado en todos los procesos de forma tal que no podamos acceder a su código para poder pasar a modo kernel, es decir, cambiar de ring. De esta forma, cambiar a modo kernel es simplemente cambiar de ring y a partir de entonces voy a poder acceder a su código. KERNBASE está en esa dirección específica por comodidad porque si está al final del espacio de direcciones yo sé que en todo proceso de usuario va a estar mapeado ahí por lo que no me voy a tener que acordar que en el medio del espacio de direcciones virtuales está el kernel.

¿Por qué los arreglos envs y pages están mapeados en el espacio de direcciones virtuales de los procesos de usuario? JOS es un exokernel por lo que hace lo mínimo e indispensable y delega mucha funcionalidad a la capa de arriba, que es la librería estándar de JOS. Entonces va a haber funcionalidades que vamos a implementar en modo usuario y para esto, el modo usuario va a necesitar acceder a los procesos y a las páginas para poder leerlas y hacer cosas con eso, pero no va a poder modificarlas directamente.

env_setup_vm()
Vamos a tener que crear un page directory nuevo y crearle los mapeos necesarios para que esté todo mapeado.
Si el directorio de páginas del kernel ya tiene hechos estos mapeos, ¿no podemos crear una copia del page directory del kernel? Podemos crear uno nuevo pero crearlo como una copia del page directory del kernel. Faltaría agregar el código, stack del usuario pero no lo va a hacer env_setup_vm(). Esto se hace en la tarea 2 que se encarga de la carga en memoria física del código y datos del usuario y el mapeo a su espacio de direcciones virtuales.

Cosas que tiene que tener mapeado el nuevo page directory
- Kernel (KERNTOP - KERNBASE)
- Arreglos pages y env (UVPT - UTOP)
Entonces tenemos que configurar la virtual memoria como una copia del page directory del kernel. Nunca modificamos cr3, ya que tenemos pgdir_walk o page_insert que nos permite insertar páginas a cualquier page directory

Tarea 2 - Carga de binarios

Nos falta el código propio del usuario para terminar de armar el espacio de direcciones. No vamos a tener heap porque en JOS no vamos a implementar sbrk() para pedir más páginas. Pero si vamos a tener data (donde vamos a tener las variables estáticas), code (donde está el código que ejecutamos) y stack.
¿De dónde sacamos data y code? Tienen que salir de algún lado pues el kernel no puede crear esporádicamente código de usuario.
¿De dónde sale el código en Linux por ejemplo? ¿en qué momento se carga el código de un proceso nuevo? Con el exec() al que le pasamos el path de un archivo. El exec() lo que hace es cargar el proceso en memoria. En JOS esto significa hacer los mapeos que hagan falta para que el código que está en ese archivo se mapee a memoria. Esto para code y data se ve claro. Por su parte, el stack no viene dentro del binario, siempre se inicializa y hay que poner el stack pointer en esa posición (USTACKTOP). Tenemos que allocar las páginas por debajo de USTACKTOP para el usuario y después tenemos que generar todos los mapeos necesarios para cargarle el código y el usuario.
Por lo que necesitamos un binario, es decir, un archivo compilado mínimo de carga. Otra cosa que necesitamos es un formato o una convención para saber dónde cargarlo. Esto es porque un binario espera estar mapeado en cierta posición de memoria virtual.
Entonces tenemos un binario que queremos cargar y correr en memoria. Queremos saber en dónde cargarlo en memoria. La paginación ya está activada en este momento. ¿Dónde ya teníamos este problema? En el bootloader de JOS. Teníamos la imagen del kernel y queríamos cargarlo en un lado tal que sea el que el kernel quiere o fue preparado para correr (recordar que había una posición KERNBASE y el kernel estaba preparado para

correr en posiciones altas de memoria). ¿De dónde sacaba esa data? Leyendo del header del binario específico de tipo elf.

Formato ELF
Tiene mucha información no solo el código y los datos. Tiene metadata dividida en secciones del binario que indica en qué dirección fue enlazada, en qué dirección espera ser cargada cada una de sus secciones y también tiene cosas como el entry point, es decir, dónde poner el stack pointer una vez cargado en memoria. Tiene program headers para saber dónde el programa se ejecuta y se carga en memoria y section headers para reinterpretar los datos y saber cómo enlazarlos.
Nos importa los program headers y el entry pointer porque leyendo y parseando este elf podemos saber qué secciones hay (.text, .data) y además en qué regiones mapearlas.

¿Cómo funciona el formato elf? Tiene un header que es un puntero a los program headers y tenemos un tabla de program headers. Cada program header indica que existe un segmento en este binario que empieza en offset y tiene un tamaño de x bytes y esa región del binario espera ser mapeada a las direcciones virtuales. El mismo program header indica dónde está dentro del binario el código y en qué direcciones espera ser mapeado. En algunos casos también dice la dirección física en la que espera ser mapeado pero esto no nos interesa.

```
struct Elf {
   uint32_t e_magic;    // must equal ELF_MAGIC //siempre tiene que ser
ELF_MAGIC si no panic
   uint8_t e_elf[12]; //
   uint16_t e_type;
   uint16_t e_machine; // para que tipo de machine funciona
   uint32_t e_version; // version del elf
   uint32_t e_entry;  //entry point
    uint32_t  e_phoff; //indica la cantidad de bytes que hay que dejar
pasar desde el comienzo del ELF para llegar al comienzo de la lista de
ph, el rpimero arranca el elf+e_phoff
   uint32_t e_shoff;
   uint32_t e_flags;
   uint16_t e_ehsize;
   uint16_t e_phentsize;
   uint16_t e_phnum;
   uint16_t e_shentsize;
   uint16_t e_shnum;
   uint16_t e_shstrndx;
};
```

```
struct Proghdr {
   uint32_t p_type;
   uint32_t p_offset; //cuanto hay que sumarle para ir al proximo ph
```

```
   uint32_t p_va; //dir virtual
   uint32_t p_pa; // dir fisica
    uint32_t p_filesz; // tamaño en bytes de los segmentos en la arc
imagen
   uint32_t p_memsz; // tamaño en bytes de los segmentos en memoria
   uint32_t p_flags;
   uint32_t p_align; //0 no esta alineado, 1 esta alineado
};
```

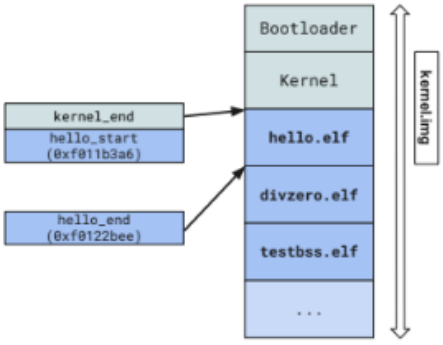
```
struct Trapframe {
   struct PushRegs tf_regs;
   uint16_t tf_es;
   uint16_t tf_padding1;
   uint16_t tf_ds;
   uint16_t tf_padding2;
   uint32_t tf_trapno;
   /* below here defined by x86 hardware */
   uint32_t tf_err; // código de error
   uintptr_t tf_eip; //instruction pointer
   uint16_t tf_cs;  //code segment
   uint16_t tf_padding3;
   uint32_t tf_eflags;
   /* below here only when crossing rings, such as from user to kernel
*/
    uintptr_t tf_esp; //stack pointer, el del usuario. Se guarda el
valor y se cambia para pasar al del kernel
   uint16_t tf_ss;
   uint16_t tf_padding4;
}
```

filesize → cuántos bytes va a copiar del archivo
memsize → tamaño final en memoria que tiene que tener esa región
Están separados porque son campos distintos. Se puede dar el caso en que memsize es más grande que filesize. Esta es una forma de optimizar espacio porque la diferencia entre memsize y filesize se rellena con ceros. Si es menor algo salió mal. Usualmente son iguales.

Tenemos 2 inconvenientes para cargar el binario: necesitamos un archivo que cargar y una forma de cargarlo. ELF es la convención pero ¿de dónde leemos el ELF si no tenemos un sistema de archivos en JOS todavía? Dado que no contamos con un sistema de archivos vamos a incluir los procesos de usuario dentro de la imagen del kernel así están accesibles como una región de memoria. Si tuviéramos un sistema de archivos, el binario lo sacaríamos de disco. Pero no lo tenemos porque no lo implementamos aún. Es un desperdicio pero funciona.

Los programas de usuario
En la carpeta del esqueleto user están los programas de usuario de JOS. Algunos son buenos, otros quieren romper JOS. Todos estos archivos están en formato ELF, se van a poder correr y tienen como fin testear. Cada uno de estos programas se puede compilar por separado. Nos interesan las regiones de tipo LOAD (ejecutables).
¿Cómo creo un proceso de usuario si no tenemos syscalls todavía? En el kernel, después de todas las inicializaciones, se llama a una macro ENV_CREATE que crea el proceso de usuario. Es decir, internamente en el kernel estamos creando un proceso de usuario. El usuario no va a poder crear sus propios procesos porque no existe fork() todavía, se tiene que crear en el kernel. Esta es una de las limitaciones que vamos a tener en el TP2. En el TP3 vamos a incluir fork().

Imagen del kernel:



Inicialmente teníamos al bootloader y al kernel que venían como en el mismo paquete. Ahora, además de eso, se agregan todos los ejecutables de todos los programas que agregamos. Como el bootloader carga toda la imagen, todo esto se va a cargar en memoria inicialmente y el símbolo env va a estar al final de esto. Con lo cual, de cierta forma tenemos a todos los binarios mapeados en memoria y podemos interpretar cada una de estas regiones de memoria como un struct Env (formato elf). Todo esto ocurre en el momento de enlazado. El proceso de enlazado nos va a dejar símbolos que indican dónde empieza y termina cada una de estas regiones. Estas son direcciones de memoria virtuales ya que el kernel está mapeado.

env_create()
Recibe un puntero a un binario que va a ser la dirección de estos símbolos de algún lado de la imagen del kernel en donde están mapeados los procesos de usuario. Esta es la forma de mapear procesos de usuario porque aún no tenemos disco.

Entonces ya tenemos las respuestas a las preguntas:

1) ¿Cómo obtenemos los binarios que vamos a cargar? Están mapeados hardcodeados en la imagen del kernel y tenemos estos símbolos para encontrarlos. La macro ENV_CREATE nos permite abstraernos de esto y referirnos solamente con el nombre.
2) Si ya tenemos la imagen, ¿cómo la cargamos? Está el formato ELF. Al interpretarlo nos dice en qué regiones tenemos que mapear y dónde hay que mapearlas.

Esto es lo que vamos hacer

region_alloc()

Hace llamadas a page_alloc() y page_insert() para mapear una región continua de binario. Recibe una longitud y un tamaño y hay que realizar las llamadas que hagan falta. Esto se hace en el page directory del environment. Esta función es auxiliar, no carga ningún código en ningún lado.

load_icode()

Es la función que va a cargar el proceso en memoria. Tenemos que usar region_alloc(), encontrar los program headers, mapearlos en las regiones apropiadas, inicializar el USTACK y configurar el entry pointer en el punto de entrada que diga ELF.

Recibe un environment y un binario. Es similar a lo que hace el bootloader excepto que en lugar de leer de disco vamos a sacar los datos de memoria.

```
// Hints:
   //  Load each program segment into virtual memory
   //  at the address specified in the ELF section header.
   //  You should only load segments with ph->p_type == ELF_PROG_LOAD.
   //  Each segment's virtual address can be found in ph->p_va
   //  and its size in memory can be found in ph->p_memsz.
   //  The ph->p_filesz bytes from the ELF binary, starting at
   //  'binary + ph->p_offset', should be copied to virtual address
   //  ph->p_va.  Any remaining memory bytes should be cleared to zero.
   //  (The ELF header should have ph->p_filesz <= ph->p_memsz.)
   //  Use functions from the previous lab to allocate and map pages.
   //
   //  All page protection bits should be user read/write for now.
   //  ELF segments are not necessarily page-aligned, but you can
   //  assume for this function that no two segments will touch
   //  the same virtual page.
   //
   //  You may find a function like region_alloc useful.
   //
   //  Loading the segments is much simpler if you can move data
   //  directly into the virtual addresses stored in the ELF binary.
   //  So which page directory should be in force during
   //  this function?
   //
   //  You must also do something with the program's entry point,
   //  to make sure that the environment starts executing there.
   //  What?  (See env_run() and env_pop_tf() below.)
```

Tarea 3 - Ejecución de un proceso

¿Qué cosas tienen que pasar para poder decir que estamos ejecutando un proceso? Pasar al address space del proceso, Hacer un context switch para pasar del ring 0 al ring 3, tenemos que asegurarnos que el espacio de direcciones virtuales del proceso de usuario esté siendo usado (es decir, cambiar cr3), restaurar todos los registros del CPU para que estén tal cual el proceso los había dejado (en este caso, como el proceso es nuevo tenemos que ponerlos tal cual los inicializamos. Van a estar la mayoría en cero salvo el stack y el instruction pointer que van a estar inicializados en los valores del inicio del stack y lo que dijo el elf). Tenemos que ver cómo implementamos esto.

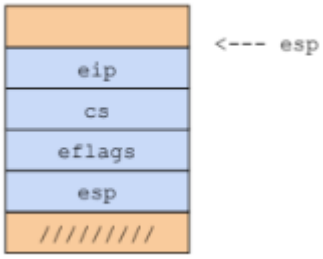
Después tenemos otras preguntas porque estamos pasando a modo usuario. Estas son: ¿cómo volvemos?, ¿cómo hacemos para que ese proceso no rompa la memoria del kernel?, ¿cómo hacemos para que ese proceso no monopolice la CPU y no nos lo devuelva?, ¿cómo permitimos una syscall?

El primer paso es cómo ejecutar un proceso que se reduce a cómo hacer un context switch (es decir, cómo cambiar el entorno que está corriendo).

Cambio de contexto
Cuando hablamos de context switch estamos hablando de congelar el estado del CPU, guardar ese estado en un lado y poner el estado del CPU en otra cosa puede ser del proceso o del kernel.
En este caso tenemos kernel y proceso de usuario. Hay 2 tipos de context switch que pueden ocurrir: de kernel a user (ejemplo: el kernel ejecuta un proceso y lo pone en memoria) y de user a kernel (ejemplo: una syscall o una interrupción). Requerimos soporte del hardware para esto, el software únicamente no lo podría hacer.
Una syscall es una interrupción de x86 por lo que la única forma de volver de modo usuario a modo kernel es a través de una interrupción. A nivel de hardware, el mecanismo de interrupción se usa para hacer ese paso. Es el único mecanismo que permite decrementar el ring en el que estás (en x86).
En contraparte, para pasar de modo kernel a modo usuario se usa la instrucción privilegiada **iret**.
¿Qué hace iret? Vayamos un paso antes, ¿qué hace ret? Lee del stack el primer valor y lo carga en el instruction pointer. Call deja en el stack el instruction pointer y salta, ret lee del stack el instruction pointer y vuelve. Por lo tanto, son contraparte.

Los interrumps iret funcionan similar. Cuando se genera una interrupción, el hardware va a dejar dentro del stack del kernel parte del contexto (code segment, instruction pointer, flags, stack pointer). Iret va a leer esos valores del stack y los va a cargar en el CPU como parte de una misma operación. A través de este mecanismo, estamos permitiendo cambiar simultáneamente el code segment, el stack pointer y el instruction pointer.

Iret espera que el stack se vea así:



Como es una instrucción privilegiada es una forma de modificar code segment y eflags que de otra forma no lo podríamos hacer.
Sin embargo, esta instrucción sólo cubre unos pocos registros, por lo que tenemos que restaurar el resto antes.

env_pop_tf()
Apunta el stack pointer al struct Trapframe que le paso por parámetro. No me importa preocuparme por el stack anterior porque es un cambio de contexto. Todavía nos falta restaurar cr3. Para esto está env_run().

env_run()
Agarra un environment y lo manda a correr al CPU. Es tarea de mantenimiento.

Hasta la parte 3 vamos a poder ejecutar procesos de usuario. Vamos a ver cómo configurar las instrucciones para permitir syscalls.

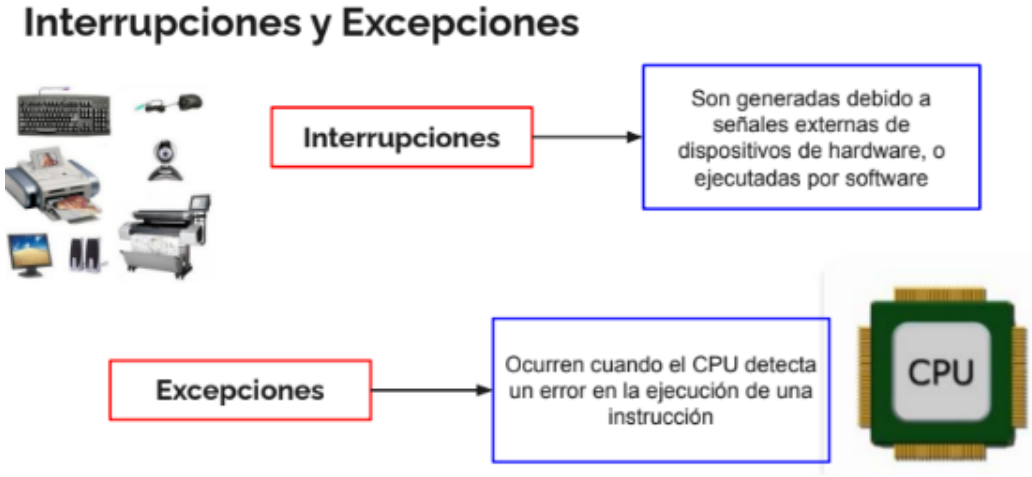
Clase práctica: TP2 Parte 1

Parte 4

Interrupciones en x86

Interrupciones y excepciones

¿Qué son las interrupciones en x86? Básicamente x86 maneja 2 conceptos: interrupciones y excepciones. No hay una estandarización de la nomenclatura.



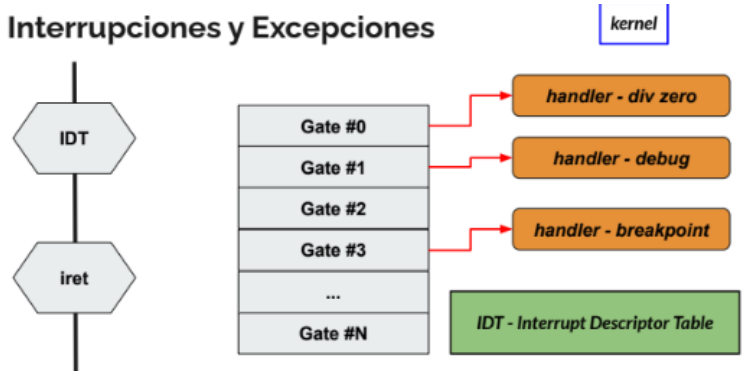
Una forma de generar la syscall es por medio de una interrupción. Hay una instrucción en assembly que se llama int a la que se le pasa el número de excepción a generar. Por otro lado, a las excepciones las genera la CPU cuando hay una violación a una regla de negocio con el SO. Son ejemplos de excepciones: división por cero, page fault. Se denominan terminales porque causan que el proceso en cuestión aborte la ejecución.

Flujo conceptual: tenemos un programa que es una abstracción de un código que está siendo ejecutado en otra abstracción que es el proceso provisto por el kernel y se genera una excepción/interrupción por software o una interrupción por hardware.
IDT → Interrupt Descriptor Table. Es una tabla de descriptores que guarda información de funciones (handlers) de diferentes interrupciones. Es como un diccionario indexado por el número de interrupción.

Tenemos implementado el proceso de handler-iret-programa. Vamos a implementar el proceso de programa-IDT-handler.
Vamos a configurar dónde va a estar ubicada la IDT, cuáles van a ser los handlers, qué código van a tener y qué interrupciones se van a poder generar desde el usuario.

IDT

Hay una cantidad finita de entradas. Algunas tienen un nombre propio. El resto son definidas por el usuario (el que desarrolla el kernel).



Es un zoom en el kernel.

IDT

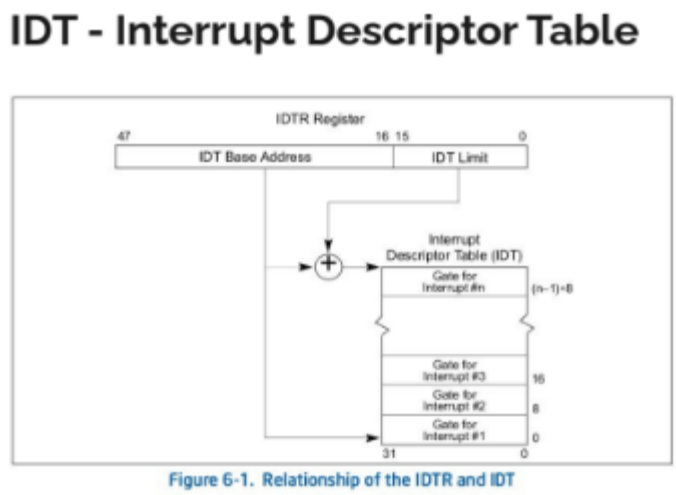


Figure 6-1. Relationship of the IDTR and IDT

Este es el registro de la tabla y dónde se la puede encontrar. Es parecida a la tabla global de descriptores. Tiene un registro para acceder a la tabla.Cada entrada tiene metadata acerca del handler y un puntero a ese handler. El kernel destina un lugar en el binario para describir los handlers que son funciones. Cuando se produce una excepción, la CPU viene a esta tabla, se fija cuál es el número de esa excepción y ejecuta el handler.
Antes de esto hay un preparativo que la CPU hace para dejarle un poco de contexto para el que viene después que en este caso es el kernel. Este contexto son valores de registros importantes como para que el kernel tenga información, agregue información adicional y resuelva las excepciones.

IDT - Interrupt Descriptors
Cada descriptor tiene la siguiente forma:

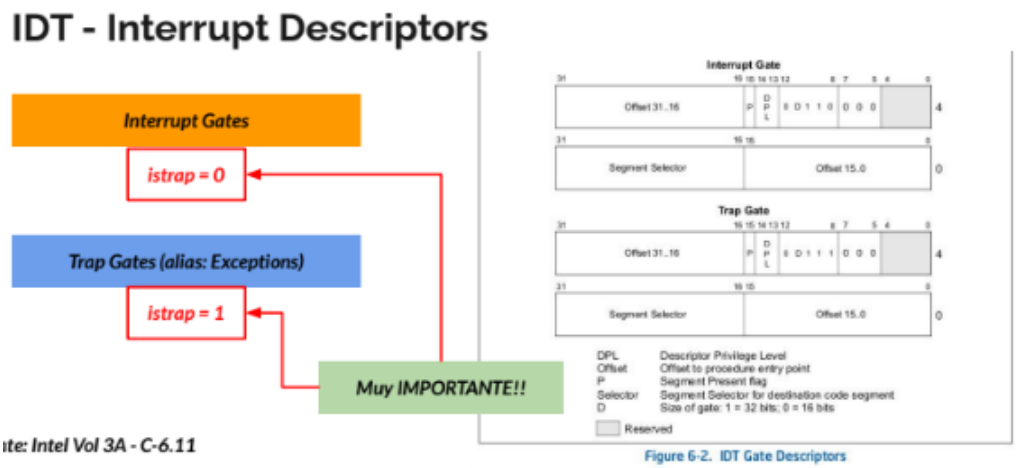


Figure 6-2. IDT Gate Descriptors

offset → es la dirección a memoria en donde está ese handler
El resto es metadata que representa cuál es el comportamiento que tiene que tener ese descriptor. Por ejemplo, el istrap. Es un bit por lo que puede estar en cero o en uno. Es importante porque es causante de muchos problemas en este TP.

La diferencia entre interrupt gates y trap gates está en sí se permiten (istrap=1) o no (istrap=0) anidaciones de interrupciones. Esto es análogo a permitir que una función pueda llamar a otra función hasta que finalmente se retorna algún valor y se vayan cerrando las funciones. Si esto no se permitiera las funciones deberían ser secuenciales a un mismo nivel.
Cuando se permiten las interrupciones anidadas hay una priorización. Es una funcionalidad que cualquier arquitectura quisiera tener.
Cuando istrap = 0 y me llagan varias interrupciones, la CPU las va a encolando y las resuelve por orden de llegada.

Vamos a implementar el caso en que no se permiten interrupciones anidadas. Permitirlas resulta más complejo.

IDT - Interrupt Call Flow
Acá tenemos lo que es el flujo de cada una de las interrupciones:

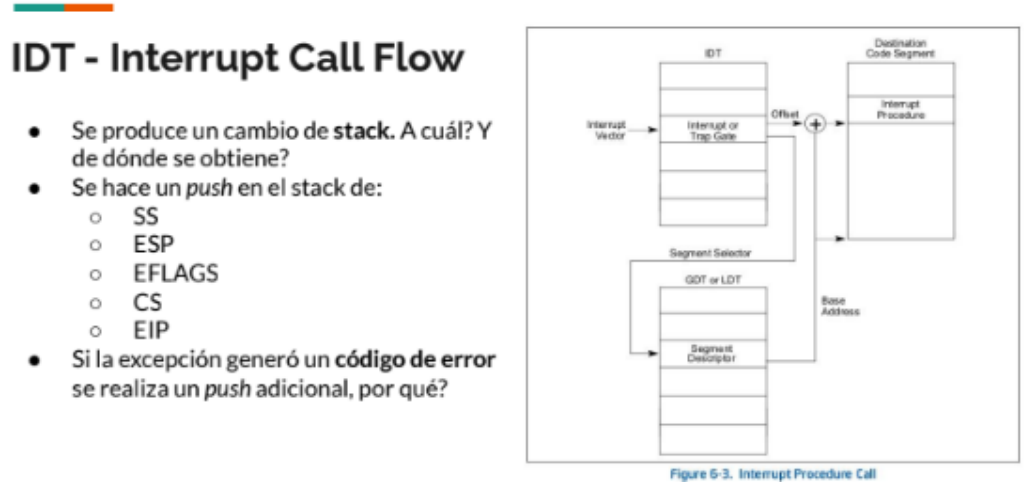


Figure 6-3. Interrupt Procedure Call

Algo importante a considerar es que cada interrupción va a tener asociada una función que va a tener su propio stack que no puede ser el del usuario por lo que tiene que saltar a uno. Hay que configurar esto. Cuando ocurre una interrupción/excepción la CPU busca un stack para pushear el contexto pues este se tiene que guardar en un lugar que es el stack por decisión de la arquitectura.
Siempre se van a pushear: SS, ESP, EFLAGS, CS y EIP. Pero si se genera un código de error, se realiza un push adicional.

Podemos ver el stack antes y después de que se produzca la interrupción/excepción:

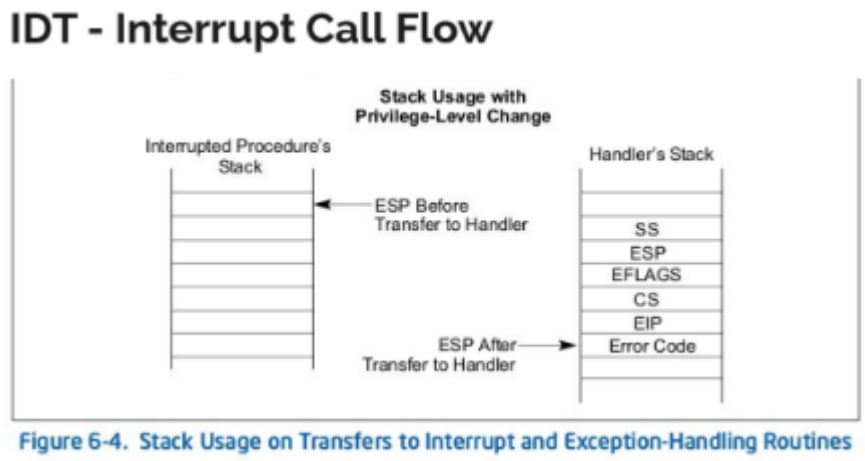


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

El kernel no sabe si se pushea un código de error a priori.
No tiene sentido que el stack del handler sea el del usuario, va a tener que ver con el kernel. El tema es dónde se configura el mismo y cómo sabe la CPU que está ahí.
No hay un stack por proceso, hay uno por core. Si tuviéramos un stack por proceso se podrían implementar interrupciones anidadas.

Si bien todos los handlers van a recibir un error code, en el struct de Trapframe va a haber un campo para este error así no se necesitan distintas estructuras y tipos de datos para distintos tipos de handlers. Los handlers que no usan el error code reciben un cero en esta variable y no la van a usar. La CPU se encarga de pushear los registros que comentamos, es tarea del kernel pushear otros registros y encargarse del error code.

Información de los handlers:

Table 6-1. Protected-Mode Exceptions and Interrupts

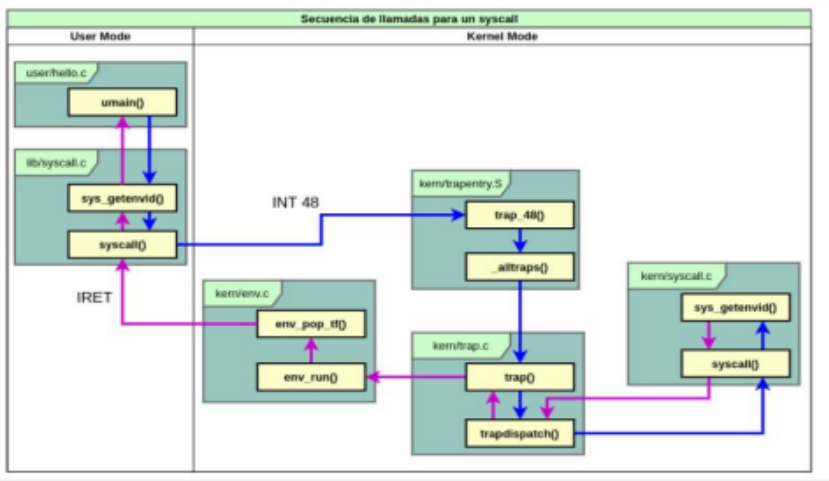
Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 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	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

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

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.[2]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[3]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[4]
20	#VE	Virtualization Exception	Fault	No	EPT violations[5]
21-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

Ignorar la columna type. Indica cómo va a actuar la CPU en caso que ocurra un falla. El trap en 0 o 1 no tiene nada que ver con esta columna. Otra cosa que define esta columna es cómo se debería comportar el handler de la excepción.

Están todas las llamadas que se van haciendo.



kern_syscalls

- sys_cputs: Estamos parado desde modo usuario, si se intenta acceder a una memoria que no esta mapeada el programa tiene que morir. Lo que deseamos es que el kernel corrobore si el puntero a la region que se intenta acceder este mapeado o sea valido, el kernel tiene q tener protecciones, el usuario no puede acceder a cosas del kernel
  Ejemplos de errores:
  Intenta acceder a una dirección del kernel

```
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    sys_cputs((char*)1, 1);
}
```

  Define bien la variable, pero el tamaño es excesivo y no esta mapeado todo ese tamanio

```
#include <inc/lib.h>

const char *hello = "hello, world\n";

void
umain(int argc, char **argv)
{
    sys_cputs(hello, 1024*1024);
}
```

- Le pedimos al kernel que imprima su código, desde el usuario esto no es posible ya que solo tiene permisos el kernel en leer el kernbase

```
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    // try to print the kernel entry point
    sys_cputs((char*)0xf010000c, 100);
}
```

Parte 5

Es para que no se pueda llamar a cualquier syscall desde el usuario.
Implementar el chequeo para que todo puntero a memoria que nos pase el usuario tenemos que revisarlo, si está mapeado en el espacio del kernel del usuario y además tiene los permisos que requiere la syscall (si requiere write que sea solo permiso de write). hay una función, user_mem_check()
user mem _assert() utiliza user_mem_check()
Recibe un usuario (env), dirección virtual (comienzo del rango), longitud del rango y los permisos. Simplemente es preguntar si el env en ese rango tiene los permisos. Si es true devuelve 0, si es false devuelve -E_FAULT

tenemos el page directory, entonces a sus page table y tenemos el page_dir_walk
Las páginas no están alineadas, hay que hacer un redondeo para poder correr el rango completo. La variable global se utiliza para almacenar en caso de que no tenga permisos la primera dirección de memoria de ese rango que generó el error
Chequear que la dirección este por debajo de ULIM, nunca accedemos a direcciones del kernel (primer chequeo)

panic en ring 0 del trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.

    // Unexpected trap: The user process or the kernel has a bug.
    print_trapframe(tf);
    if (tf->tf_cs == GD_KT)
        panic("unhandled trap in kernel");
    else {
        env_destroy(curenv);
        return;
    }
}
```

agregar un handler para page_fault hay que agregar un chequeo en dispatch con el panic del ring 0

---

user/buggyhello.c

```
void
umain(int argc, char **argv)
{
    sys_cputs((char*)1, 1);
}
```

Es un programa de usuario que intenta imprimir la dirección de memoria 1. ¿Que es lo que pasa cuando quiero ejecutar este programa de usuario? Hay un problema, y es que estamos queriendo acceder a memoria que no esta mapeada. Yo esperaría que el programa muera por intentar acceder a una memoria que no es suya.

sys_cputs→ es un print. Termina generando una desreferencia del lado del kernel. Si lo dejaramos asi, tendriamos un kernel lleno de vulnerabilidades.

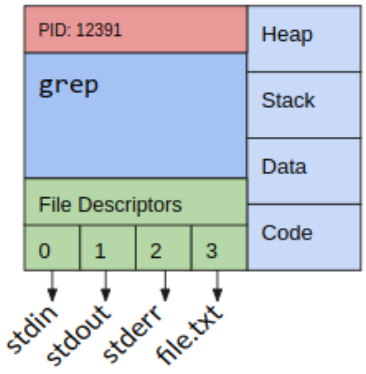
```
static void
sys_cputs(const char *s, size_t len)
{
    // Check that the user has permission to read memory [s,
    // Destroy the environment if not.

    // LAB 3: Your code here.

    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

Código

Diapositivas

Procesos en el JOS

- Los procesos en JOS se llaman **environments**
- En el TP2 se implementa:
  - Subsistema de *environments*
  - Ejecución de un **único proceso**
  - Soporte para syscalls
- En el TP3 se implementará:
  - Scheduler
  - Soporte para multiprocesador
  - Fork
  - Syscalls IPC



La cajita es todo lo que necesitamos para un proceso. Proceso es el término genérico, JOS a sus procesos se le llama environment.

- Modelados con el *struct Env*
- Toda la información de un proceso
  - ID (*env_id*)
  - Estado del CPU (*env_tf*)
  - Memoria virtual (*env_pgdir*)
  - Estado
- Cada proceso tiene su propio ***page directory***

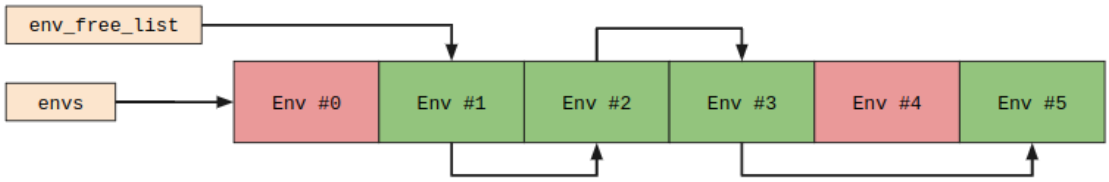
```
struct Env {
  struct Trapframe env_tf;
  struct Env *env_link;
  envid_t env_id;
  envid_t env_parent_id;
  enum EnvType env_type;
  unsigned env_status;
  uint32_t env_runs;

  // Address space
  pde_t *env_pgdir;
};
```

Este Struct representa todo lo necesario de un proceso.
De lo que vemos ahi, el env_id es el process id (el número que permite identificar al proceso). Parent_id es el nombre del proceso padre, el proceso que creo.
El trap frame,va a decodificar el estado de los registros de la cpu de un proceso, para facilitar el context switch - almacena la imagen de la CPU cuando se da el cambio de contexto- ¿por que se llama trapframe? es por la arquitectura x86.
El env_pgdir cada proceso tiene su page directory, ¿por qué? Porque un page directory codifica todo un espacio de direcciones virtuales, tener varios page directory, permite tener distintos espacios de memoria mapeados, para distintos procesos. El trapFrame env_tf lo que va a tener es una copia de los registros de propósito general.

Process Control Block

- El arreglo *envs* contiene **todos los procesos**
  - Similar a *pages*
  - Lista enlazada de "procesos libres"
- Variable *curenv* contiene al proceso actual

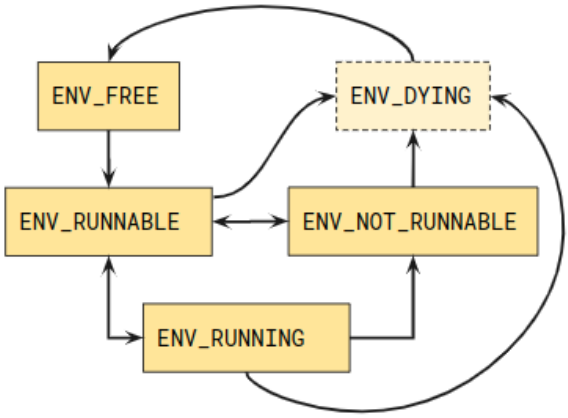
```
#define NENV ....
struct Env *envs;
struct Env *curenv
static struct Env *env_free_list;
```



Es una estructura en el kernel de los procesos que el kernel conoce que está corriendo. Este arreglo de procesos se llama envs, este arreglo tiene 1024 environments, donde el índice dentro de cada env va a corresponderse con su env_id.
Además, el env_free_list tendrá las posiciones libres dentro de envs. ¿Que es una posición libre y una ocupada? En cada pos hay un bloque de los de arriba, que esté ocupado significa que existe un proceso cuyo process_id es 0 que está en mi memoria del kernel, quiere decir que estoy guardando data de un proceso, **independientemente** de si se está ejecutando **en este momento** o no. Cuando todas las posiciones de envs están ocupadas, el kernel ya no puede crear más procesos.
curenv → es el puntero al proceso del arreglo envs que está ejecutando justamente en ese momento se esté ejecutando en la cpu.
NENV = 1024 = 2^{10}

Procesos en el JOS - Estados

- Los procesos en JOS tienen 5 estados posibles
  - **Free**: el struct Env está en la lista de procesos libres
  - **Runnable**: el proceso está listo para ser ejecutado en la CPU (hasta que el scheduler lo elija)
  - **Not runnable**: bloqueado esperando alguna operación. No elegible por el scheduler
  - **Running**: actualmente en el CPU
  - **Dying**: estado especial *opcional* si un proceso es terminado mientras estaba corriendo *en otro CPU*



¿Que estados vamos a manejar en JOS?
ENV_FREE = son los verdes del array envs
ENV_RUNNABLE= "no estoy corriendo en la cpu, pero quiero correrme en el cpu", esta a la espera de que el kernel le de lugar para correr.

ENV_RUNNING = “estoy corriendo en la cpu”, es el proceso actual activo en la cpu
ENV_NOT_RUNNABLE= “no estoy corriendo en la cpu y tampoco puedo correr en la cpu”, por ej porque puede estar bloqueado en una op de entrada/salida

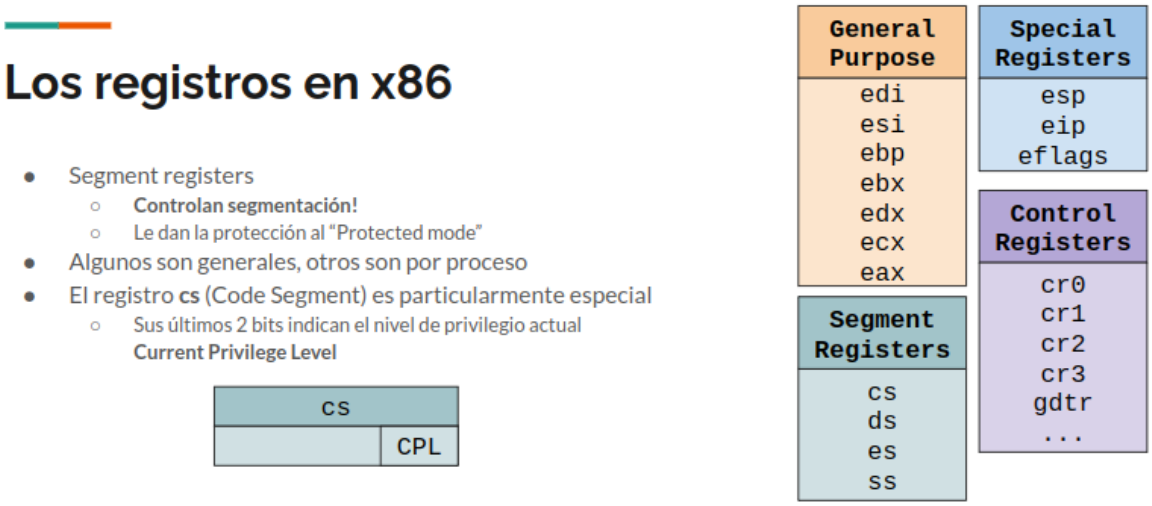
ENV_DYING = es una especie de proceso zombie, que solo tiene sentido cuando tengamos múltiples cpus.

Pasaje de RUNNABLE a NOT RUNNABLE
Opciones

- alguien te mande una señal a tu proceso que le indica al kernel que no podes correr, ocurre en gdb

Pasaje de RUNNABLE a RUNNING
Porque comienzas a ejecutarse

Pasaje de RUNNING A NOT RUNNABLE
Porque te bloqueaste con una op de entrada/salida o un pipe



El trap frame almacena los registros.
Los registros estan vinculados a su arquitectura.
x86 funciona de la siguiente manera con los registros:

- Propósito general
  - se pueden usar para lo que quieras
- Registros especiales
  - esp apunta a donde esta el stack
  - eip apunta a la direccion donde esta la instrucción que vamos a ejecutar
  - eflags son los flags
- Registros de segmentos
  - cs nos dice a qué segmento estamos accediendo cuando accedemos al código. En los últimos dos bits se almacena el CPL (el nivel de privilegio). ¿Qué quiere decir esto? en modo protegido, si vos agarras una máquina y la congelas en el tiempo, si vos vas a fijar en la cpu los últimos dos bits sabes con 100% de certeza en qué nivel de privilegio se está ejecutando la cpu en ese momento.
  - ds nos dice a qué segmento estamos accediendo cuando accedemos a data
  - es extended segmento

    - ss es el que se usa cada vez que queres acceder al stack
      son importantes porque pasan por todo lo que es la segmentación, mecanismos de segmentación orientado a permisos. La segmentación se utiliza para manejar permisos.
- Registros de control
    - cr0 registro dentro de la cpu cuyo único propósito es controlar el funcionamiento de la cpu, cada bit tiene un proposito
    - cr1 registro dentro de la cpu cuyo único propósito es controlar el funcionamiento de la cpu
    - cr2
    - cr3 almacena la dirección de memoria física donde está la tabla de paginación que va a usar en ese momento la MMU
    - gdtr → global descriptor table, dirección de memoria física donde van a estar los segmentos de memoria

Modo protegido > se protege mediante la segmentación

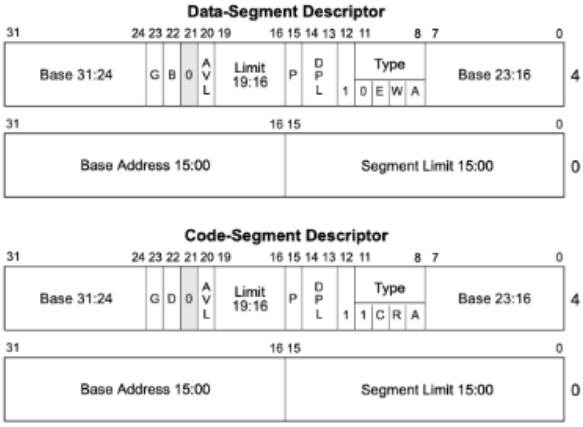
Segmentación en x86

- En **modo protegido** se tiene activada la **segmentación**
- Un **segmento** es una región de memoria
    - Se definen en una tabla de descriptores de segmentos en memoria
    - La tabla se indica al CPU a través de un registro especial (e.g. Global Descriptor Table Register)
    - Definen qué **permisos** se necesitan para usar el segmento (Descriptor Privilege Level)
- Los **registros de segmento** indican cuál entrada en la tabla usar
    - Y con qué **permisos** intentan acceder (Requested Privilege Level)

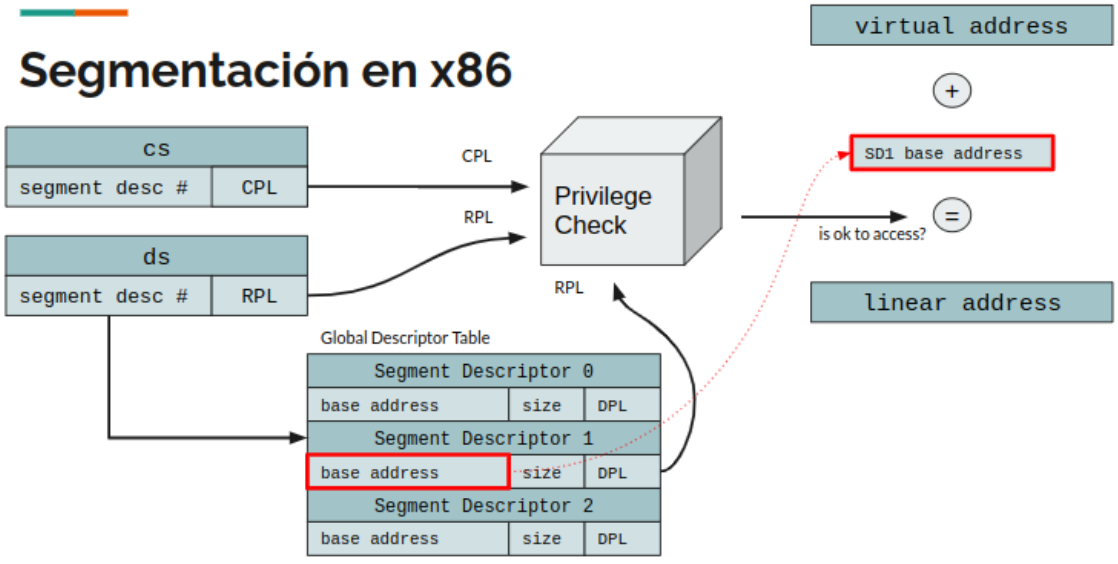
El segmento puede ser pensado como una región de memoria. Se van a definir una lista de estas regiones de memoria, donde empiezan y dónde terminan, y que nivel de privilegio tiene las mismas, es lo que se llama una tabla de descriptores de segmentos. Se acceden a través del gdtr. Cada registro de segmento tiene un número dentro que se va a usar como índice dentro de esta tabla de descriptores. El cs va a usar el segmento que describe el segmento x. Cada segmento va a tener un DPL.
Siempre se pasa por la segmentación. Recién después de esto, se pasa a paginación.
¿Por qué no nos cambia? Porque la segmentación es configurada para que la logic address sea igual a la linear address, solo en el medio se verifican los permisos.

Ejemplo de descriptor

- La **base address** indica donde comienza el segmento en memoria física
- El **segment limit** indica el tamaño del segmento
- El **Descriptor Privilege Level** (DPL) indica qué permisos tiene que tener alguien (CPL) para acceder a ese segmento en particular

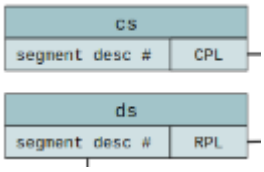


¿Cómo se ve una tabla de descriptores de segmentos?
Como se ve en la imagen, es una tabla de ese estilo. Tiene elementos que ocupan 64 bits, esos 64 bits están divididos en cosas. Usa Base and Bound. Almacena donde empieza el segmento, donde termina el segmento y que permisos requiero para que puedan acceder a este segmento. Solo quien se ejecute con un cpl igual al dpl del segmento va a poder acceder a él.



¿Qué es lo que está pasando?
Yo quiero acceder a una dirección virtual para cargar un dato, entonces voy a usar el ds, para pasar por la traducción, paso por la segmentación de esa dirección virtual que le corresponde a un dato.
Se toma el descriptor de segmento que tiene el ds y va a la tabla de descriptores globales. Se busca el segmento ahí. Tenemos la dirección base, un tamaño y el dpl. La dirección base se pone en SD1 base address y se le suma a la dirección virtual, y eso va a dar la dirección lineal. Básicamente base and bound.
El DPL va a decirme con qué privilegio puedo acceder yo a este segmento, ¿con que se compara este dpl? Se compara contra el CPL del cs, el cs siempre entra en jeugo. Lo que

importa es la caja mágica, en ella se chequean los permisos, si falla se da el segmentation fault. Si funciona, va a devolver la dirección lineal, que va a acceder a la paginación.
A cada segmento se le puede definir un tamaño, el límite son 4GB.
Esto lo digo yo JP, creo que las direcciones bases de la GDT son siempre cero,por eso son iguales la virtual y la lineal.

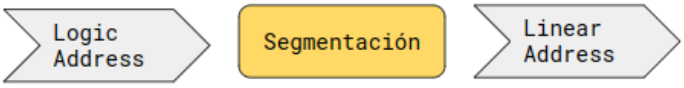


→ *el segmento desc seria GD_KT / GD_UT*

→ *el segmento desc seria GD_KD / GD_UD*

Segmentación en JOS

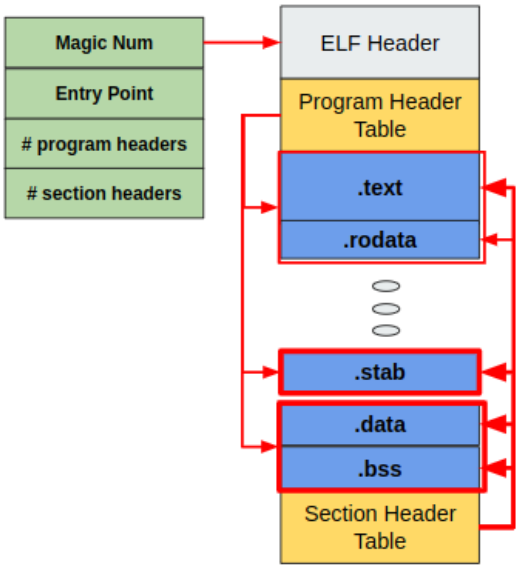
- Se usan **para manejar permisos**
  - Existen configurados **segmentos** para código y datos de usuario y kernel
- Todos tienen como **base address** 0x0 y como **limit** 4GB
  - Abarcan toda la memoria
  - Implica que la **linear address** y la **virtual address** son iguales
- La gdt está definida en código
  - Hay macros para crear descriptores de segmentos



El paso por la segmentación se usan como base como 0 y límite como 4Gb, para que abarquen toda la memoria, y para que la traducción no cambie el valor absoluto de la dirección. Lo único importante que hace es el manejo de permisos. Es lo clave de la segmentación.

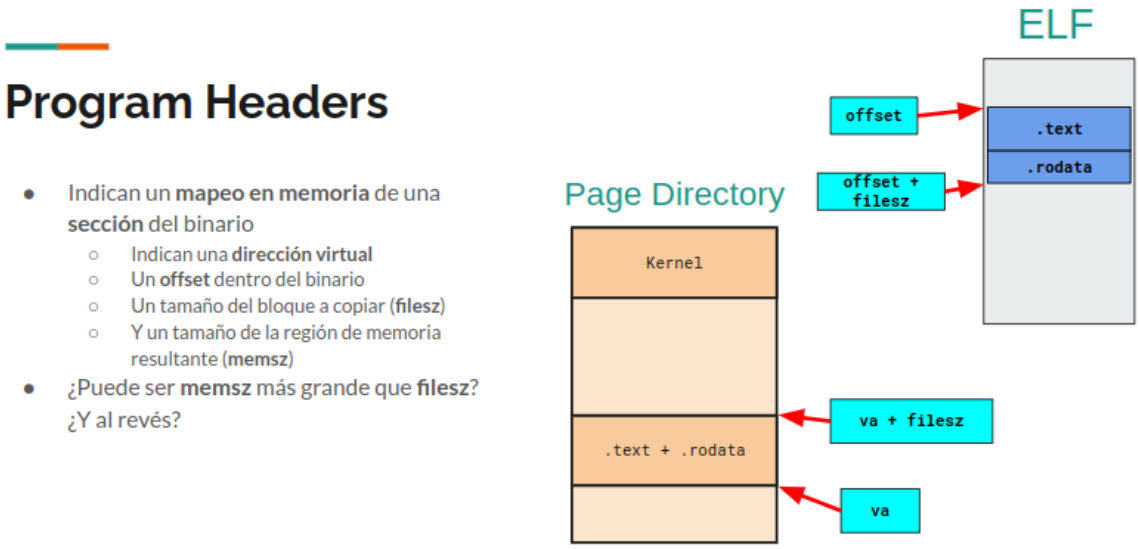
Formato ELF

- Executable and Linkable Format
- Encabezado con metadata del programa
  - Entrypoint, target arch, #headers, etc
- Conjunto de **secciones**
  - el **contenido** del binario (código, data, debug info, etc)
- Conjunto de **headers**
  - Los **program headers** indican cómo el binario debe ser **cargado en memoria**
  - Los **section headers** indican cómo el binario puede ser **enlazado**
- readelf -a obj/user/hello | less



El binario tiene el código y los datos.

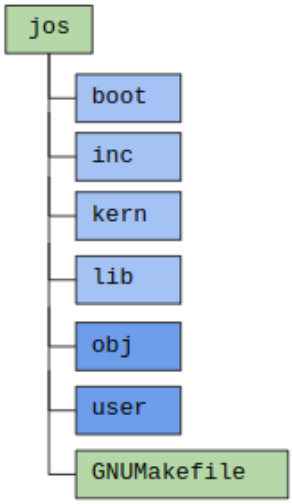
Formato ejecutable y linkeable. Header lista con metadata que nos informa que el programa para little endian big endian, este programa ocupa tantos bytes.
Este elf tiene una lista de program headers y section headers. Cada uno de estos apuntan a bloques dentro del binario que tienen código y data. Leyendo los program headers vamos a poder saber qué regiones del elf cargar y donde mapearlos en memoria virtual. Los section headers es similar, la diferencia es que los program headers se usan para cargar en memoria por ej cuando se hace un exec mientras que los section se usan para enlazar (por ej en una compilación parcial con el .o).
Si somos el kernel y encontramos en elf en disco, se puede cargar a memoria toda la data.



Cada program header nos va a decir dónde está la sección que vamos a cargar dentro del binario mismo,
ese es el offset. Desde el offset hasta el offset+fileesz es lo que vamos a tener que mapear en dirección virtual. ¿Dónde? En el espacio de direcciones virtuales de ese proceso, mapeamos esa cantidad de bytes. Desde va a va+fileesz se copia contenido del elf. Y si memsize es más grande que file size, todo el espacio vacío se llena con 0. Al reves no puede pasar, seria aumentar el tamaño del elf y cagarla.

Los programas de usuario

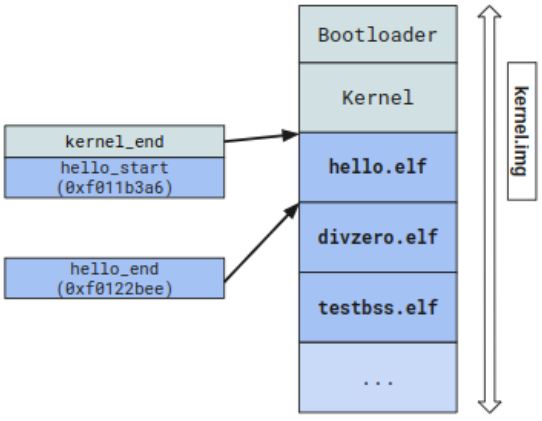
- El código de los programas de usuario está en en el directorio *user*
- Se compila en *obj/user*
- Se usa una macro para crearlos:
  - **ENV_CREATE** en *init.c*
- ¿Cómo podríamos acceder desde JOS, si aún no hay filesystem?



Como accedemos a un binario si no tenemos la nocion de archivos. La clave es no tenemos filesystem, con lo cual, la gente de JOS dijo vamos a hacer que los programas de usuario, vengan embebidos con la imagen del kernel.

Los programas de usuario

- Están **embebidos** en la imágen del kernel
  - Se enlazan como parte del mismo
- En enlazador nos deja **símbolos** al principio y fin de cada programa
- Leyendo en esas direcciones, leemos bits en formato ELF
- La macro **ENV_CREATE** reemplaza estos símbolos a partir del nombre del binario y llama a **env_create()**



La imagen del kernel es la que se muestra en la imagen. Se concatenan los programas de usuario. Se concatenan de forma tal que el enlazador nos va a generar símbolos de donde empieza y dónde termina cada programa de usuario. Es una solución horrible, porque si quiero tener otro programa de usuario, tengo que volver a compilar todo el kernel, pero si no tiene un disco, es bastante útil.
Vamos a poder acceder a las direcciones virtuales casteandolas como si fueran un elf, entonces tendremos un elf.

ENV_CREATE → la macro lo que hace es al nombre de usuario que le pasamos, le concatena cosas para convertirlo en la dirección de memoria donde el binario está cargado. Y luego llama a la funcion env_create() con esa dirección de memoria, que lo que tiene que hacer es castearlo como un elf y realizar la carga del binario.

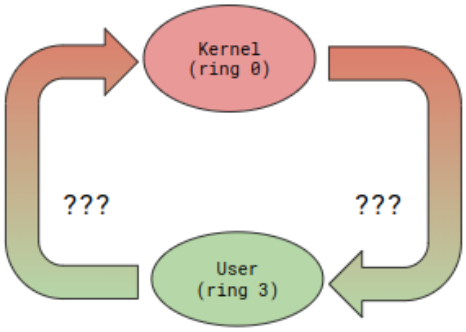
Tenemos un env listo, lo ejecutamos ¿qué es ejecutar un proceso?

- tenemos que agarrar los registros de propósito general del env, y cargarlo en los registros reales del cpu
- los registros segmento del env, y cargarlo en los registros reales del cpu. Esto implica cargar el registro cs (quien maneja los privilegios), con lo cual cargando ese registro se está cambiando al ring 3.
- modificar el cr3 para usar el page directory del proceso de usuario.

como sacamos programas maliciosos o un programa con un loop infinito → una interrupción

Cambio de contexto

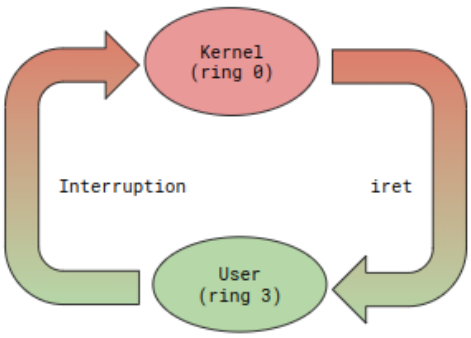
- Implica **congelar** el estado del CPU y **reemplazarlo** por otro. Puede ocurrir cambio de privilegio.
- Requiere **soporte del hardware**
- Dos tipos:
  - Kernel a User: scheduling, ejecución de un proceso
  - User a Kernel: syscall, ¿algo más?



Hay dos tipos de cambio de contexto.
Un cambio de contexto es cuando todos los registros de propósito general, los de segmento, el cr3 y el cs son congelados y reemplazados por otros set de registros. Al hacer eso, estamos cambiando cosas como el ring, los registros y el page directory.
Cuando hacemos un cambio de kernel a usuario, hacemos un cambio de contexto que pierde privilegio.
Cuando cambiamos de usuario a kernel, gana privilegio, llamando una syscall. Lo otro que lo puede generar, es un programa malicioso, queriendo entrar a un lugar de memoria no compatible con sus permisos o cuando se ejecuta durante más del tiempo esperado.

Cambio de contexto

- Kernel a User
  - Se utiliza una instrucción especial *iret* (Interrupt Return)
  - Es como *ret* pero con más propiedades
- User a Kernel
  - Se utilizan **interrupciones**
  - Funcionalidad del **hardware** configurada por el **kernel**



Modo usuario a modo kernel

La interrupción es la única forma en x86 que tenemos de ganar privilegios. Se configuran de forma tal que no puedan ser usadas

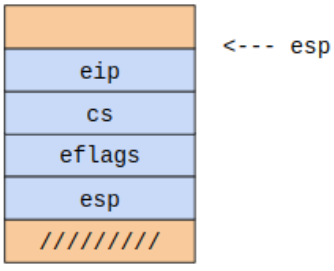
- un fallo en memoria porque hay una pagina muy mapeada, genera una interrupción y nos devuelve al kernel
- un intento de usar una instrucción no privilegiada si es de usuario, genera una interrupción y nos devuelve al kernel
- un timer es una interrupción que nos devuelve al kernel
- una syscall va a ser una interrupción que nos devuelva al kernel

Modo kernel a modo usuario
IRET básicamente deshace lo que hace la instrucción y vuelve al ring con menos privilegios. ¿Qué es lo que hace? Sirve para volver de una llamada.Call salta a una posición arbitraria de código, y deja en el stack el instruction pointer que contiene la dirección de memoria a la que tenes que volver. Entonces iret lee del stack una dirección de memoria y hace un jump a esa dirección de memoria.

La interrupción y el IRET van a funcionar de una forma similar. La interrupción la va a generar el hardware, y va a dejar el stack el estado del que acaba de salir el cpu. Ese estado, se va a usar con el iret para recuperar el estado en el que estaba, cuando vuelva al user mode.

Pasando a modo usuario - iret

- La instrucción *iret* **es privilegiada**
- Espera cierta configuración del stack:
  - Restaura los valores de **eip**, **eflags**, **cs**, y **esp** con los que encontró en el stack
- Al cambiar **eip** se salta a una sección de código arbitraria
- Al cambiar **esp** estamos cambiando el stack de llamadas
- Al cambiar **cs** se puede modificar el CPL!
  - Si se pone un 3 en el **cs** del stack se pasa a ring 3 al llamar a iret
  - ¿Podría un usuario llamar a iret teniendo un **cs** con CPL=0 en el stack? ¿Por qué?
- ¿Qué nos falta para restaurar **completamente** el estado del CPU?



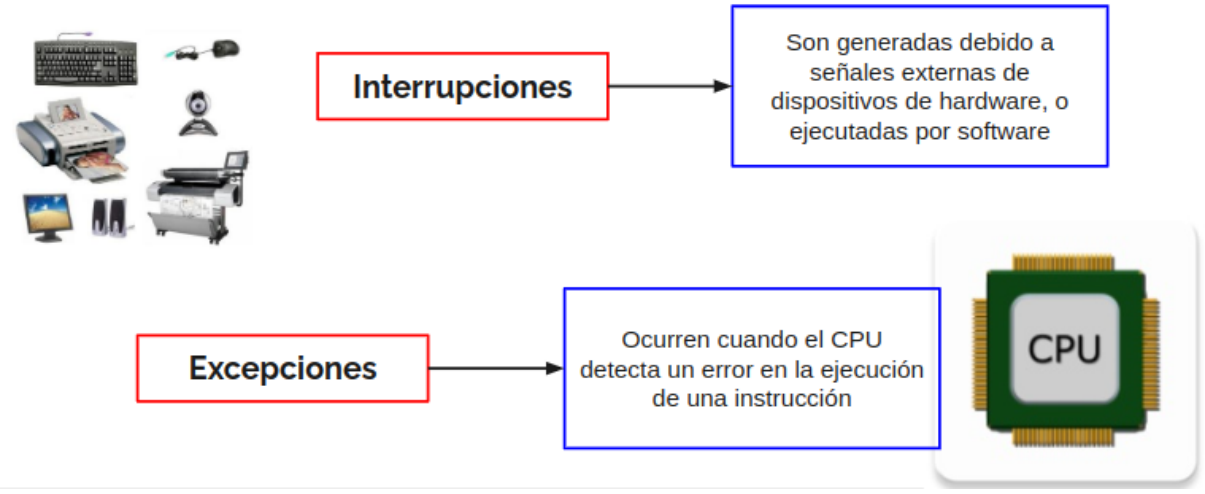
La instrucción iret es privilegiada. Solo se puede llamar desde el ring 0 (el kernel). El iret espera que el esp apunte a una pila con esos valores. ¿Por qué? Porque va a leer el eip, el cs, el eflags y el esp y lo carga en el cpu. Con eso hace el cambio de contexto. De un solo golpe, restaura el estado.
Podemos usar iret sin tener paginación activada. Antes de usarlo, se restauran los registros de propósito general y el cr3.

Procesos en el JOS - Trapframe

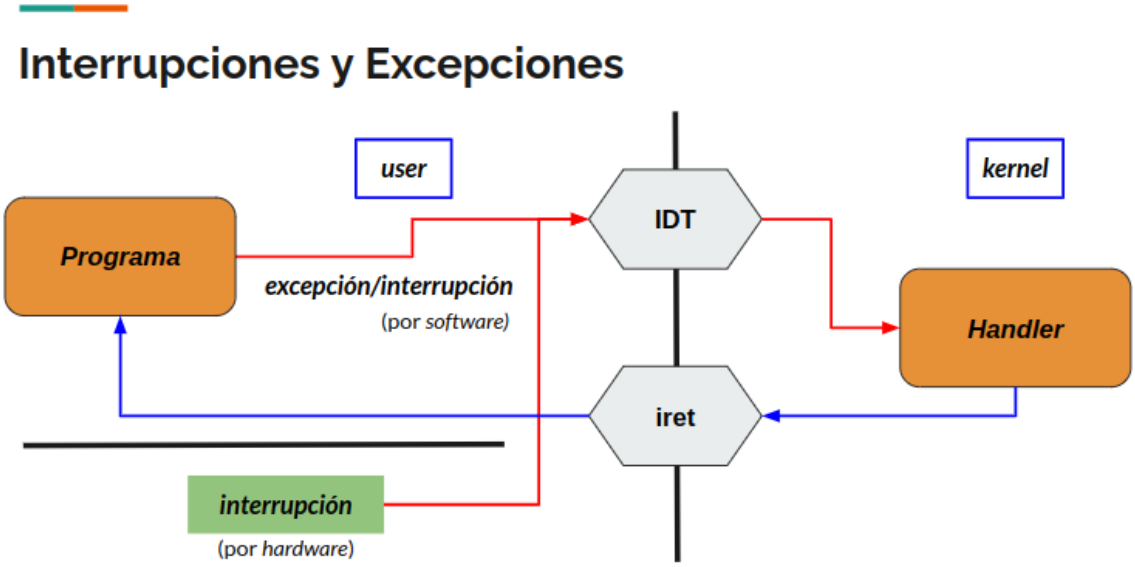
- Representa el estado de **los registros** de un environment determinado
- *PushRegs* son los registros de propósito general
  - están en el orden indicado para poder ser agregados o quitados al stack con **popal** y **pushal**
- ¿Hay un orden especial en los demás registros?

El padding es para que sea de 32 bits el cs.
El orden no es coincidencia. Está todo ideal.

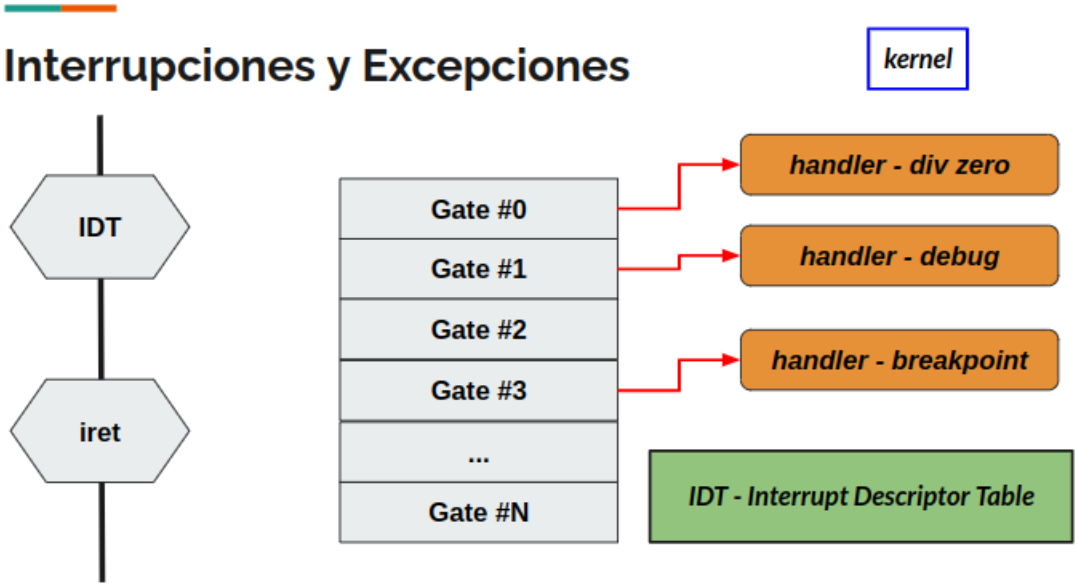
Interrupciones y Excepciones



La alu lanza excepciones.



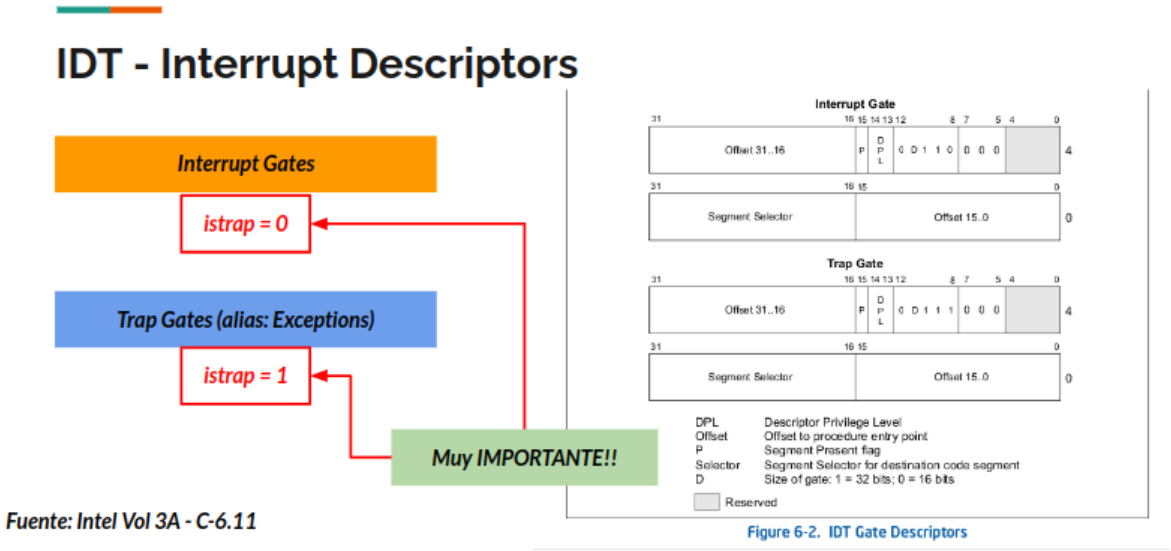
¿Qué es lo que está ocurriendo? tenemos un programa de usuario (en user land), que genera una excepción o interrupción por software o una interrupción por hardware (perifericos). Todas concluyen en una IDT, interrupt descriptor table.
Eso va llegar al kernel a través de un handler que va a manejar cada una de esas interrupciones (hay un handler por interrupción).
Cuando el kernel haya terminado de resolver cada una de las interrupciones, la tarea termina. Y retorna al proceso que la ocasionó (en caso de que se haya terminado de ejecutar o no quiera el kernel ejecutarlo, retornara a otro proceso). ¿Cómo lo va a hacer? Con IRET.



El IDT lo que tiene es con puertas, es por donde el proceso se va a poder meter y decir quiero hacer una syscall. Cada una de estas entradas va a tener un handler diferente configurado para cada una de estas interrupciones.

El kernel va a configurar la IDT.

Tiene un registro,un registro de control. El registro de control que tiene la dirección de donde se encuentra la tabla y el tamaño, cuanto ocupa la tabla. Entonces el offset se puede mover dentro de esa tabla (limitado por el tamaño) para llegar a una gate.



La distinción que se hace entre trap gate e interrupt gate.
Una interrupt gate resetea el interrupts flags, entonces no dejó pasar ninguna otra interrupción si no se manejo la actual. No permite que se aniden las interrupciones.El kernel va a manejar una detrás de la otra. Pero no para de manejar una para manejar otra.
El trap gate, se suspenden las ejecuciones de las interrupciones que se ejecutan, y comienza a ejecutarse la que llegue. Se anidan.
En el tp usamos las interrupt gate porque en JOS no se pueden anidar interrupciones, su arquitectura no lo permite.

IDT - Interrupt Call Flow

- Se produce un cambio de **stack.** ¿A cuál? Y ¿de dónde se obtiene?
- Se hace un *push* en el stack de:
  - SS
  - ESP
  - EFLAGS
  - CS
  - EIP
- Si la excepción generó un **código de error** se realiza un *push* adicional, ¿por qué?

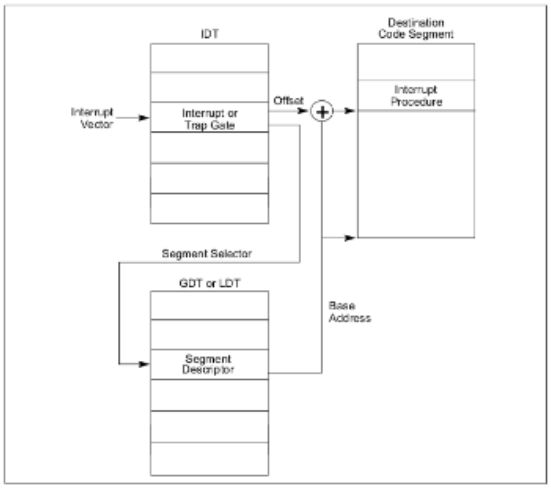


Figure 6-3. Interrupt Procedure Call

Fuente: Intel Vol 3A - C-6.12.1

Cuando se genera una interrupción, se concluye en la CPU que es quien sabe que es lo que hay que hacer.
Cuando a la CPU le llegue una interrupción, va a hacer ciertas cosas y después le va a ceder el control al kernel. Esas ciertas cosas que hace, es dejar cierta información del contexto que está pasando para que el kernel pueda manejar adecuadamente la interrupción.
Lo primero que va a pasar es que si la interrupción se generó en un ring diferente (es decir en ring3), se va a cambiar de ring (a ring 0). Y se va a hacer cambio de stack, el stack del usuario no se pisa ni se destruye (al menos en ese momento).
¿Quién es este stack? ¿Dónde está? TSS (task state segment) este segmento es uno de los posibles segmentos que se configuran en la GDT. El TSS se utiliza por default para hacer los cambios de contexto, y configurar a donde van a ir a ejecutarse los handlers, una de esas cosas que se configuran es el stack (a donde va a ir a parar el código, en que stack van a estar ejecutandose y con que nivel de privilegios).

En trap_init_percu se configura el stack del kernel.
Siempre que hay un cambio de contexto, se deben almacenar:

- SS
- ESP
- EFLAGS
- CS
- EIP

Si la excepción generó un código de error se realiza un push adicional, ¿por qué? se pushea si la interrupción se genera. ¿Cómo sabemos cuáles son las interrupciones que lo generan?La tablita de instrucciones de Intel.
PageFault es genérico (puede que la página no está, o se quiere acceder a una página sin permisos).

¿Por qué se llama trapframe? es porque en env_pop_tf la cpu pushea todos los registros esos, se genera una trap. Entonces trapframe → el marco de contexto de una trap.

La macro TRAPHANDLER pushea el número de la interrupción. La macro TRAPHANDLER_NOEC pushea un 0, y luego pushea el número de la interrupción. ¿Por qué esta diferencia? Es para que tengan el mismo formato siempre, ya que las que tienen error code pushean el propio. Entonces JOS pushea el cero en las que no generan error code, para que el formato de trapframe siempre sea el mismo.
Ambas macros llaman a _alltrap, ¿que es lo que a tener que hacer? es terminar de rellenar el trapframe. Básicamente pushea todo lo que no se ha pusheado aun.

IDT - Interrupt Call Flow

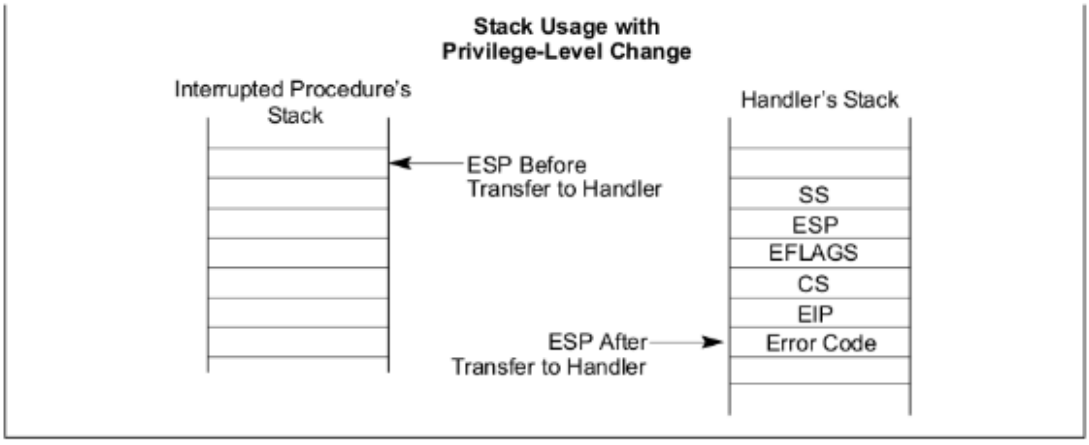


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines