# Procesos en JOS

Sistemas Operativos - FIUBA

# Integración de código con el TP2

# TP2 JOS - Introducción

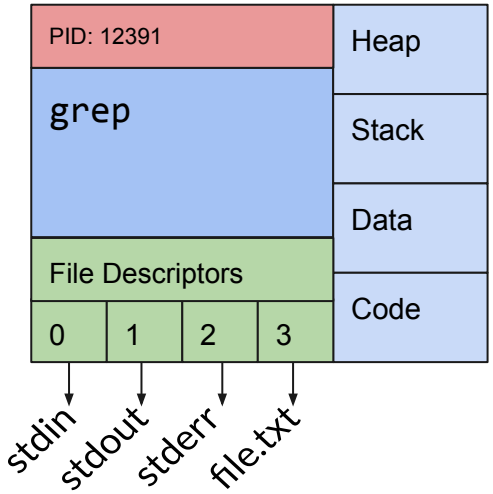
- El esqueleto para este TP está en la rama *tp2* del repo de jos de la materia
- Para integrarlo deben:
  - desde su branch *entrega_tp1* crear una nueva rama **base_tp2**
  - mergear los cambios de tp2
  - pushear base_tp2 y crear entrega_tp2
- No olvidar traer sus cambios a entrega_tp2 de las correcciones del TP1

```
// Pararse en la rama entrega_tp1
$ git checkout entrega_tp1

// Crear una nueva rama base_tp2 (y pararse en ella)
$ git checkout -b base_tp2

// Integración del "esqueleto" del tp2
// No debería haber conflictos, si los hubiera,
// resolverlos
$ git fetch --all
$ git merge catedra/tp2

// Probar que se corren las pruebas nuevas
$ make grade

// Pushear la rama base_tp2
$ git push -u origin base_tp2

// Creación de la rama entrega para el tp2 (y pararse
// en ella)
$ git checkout -b entrega_tp2
$ git push -u origin entrega_tp2
```

# Introducción a procesos (en JOS)

# 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



# Procesos en el JOS

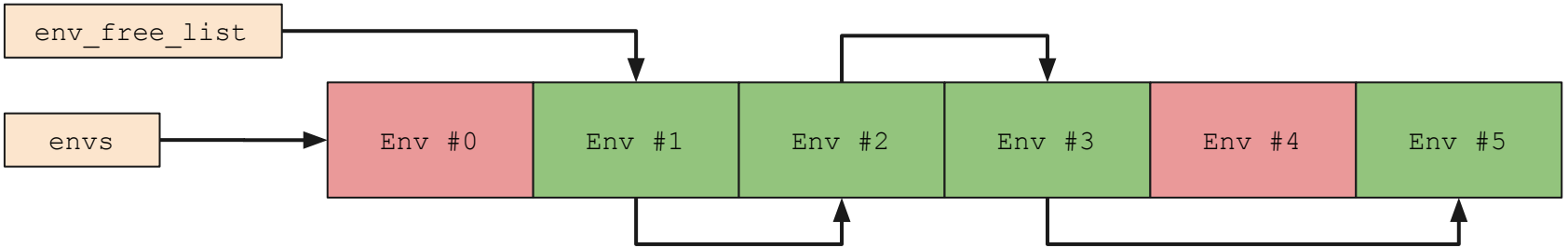
- Modelados con el *struct Env*
- Toda la información de un proceso
  - ID (*env_id*)
  - Estado del CPU (*env_tf*)
  - Memoria virtual (*env_pgdir*)
  - Estado
- Cada proceso tiene su propio ***page directory***

```
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;
};
```

# Process Control Block

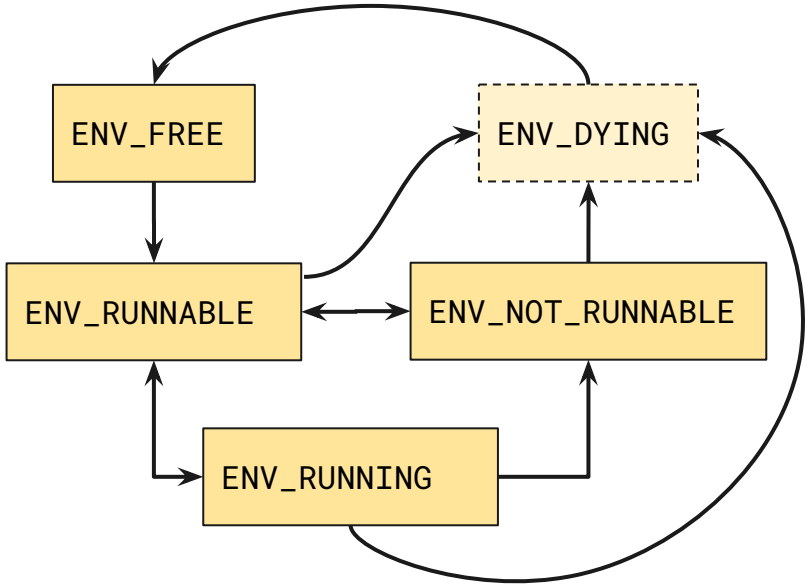
- El arreglo *envs* contiene **todos los procesos**
  - Similar a *pages*
  - Lista enlazada de “procesos libres”
- Variable *curenv* contiene al proceso actual

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



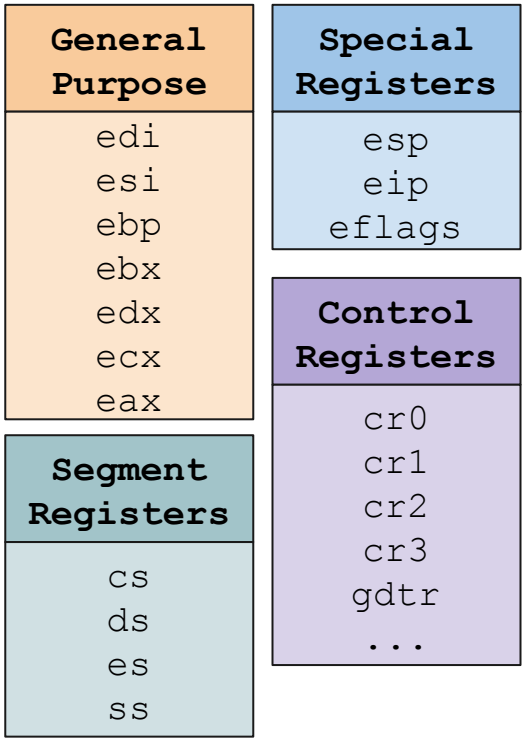
# Procesos en el JOS - Estados

- 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*



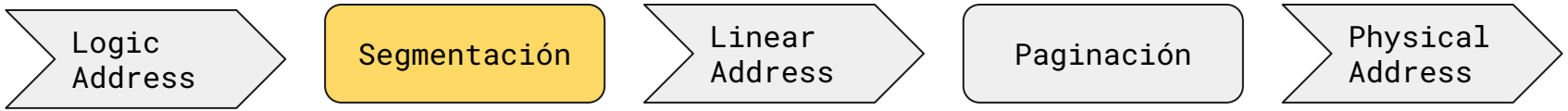
# Los registros en x86

- Segment registers
  - **Controlan segmentación!**
  - Le dan la protección al “Protected mode”
- Algunos son generales, otros son por proceso
- El registro **cs** (Code Segment) es particularmente especial
  - Sus últimos 2 bits indican el nivel de privilegio actual **Current Privilege Level**

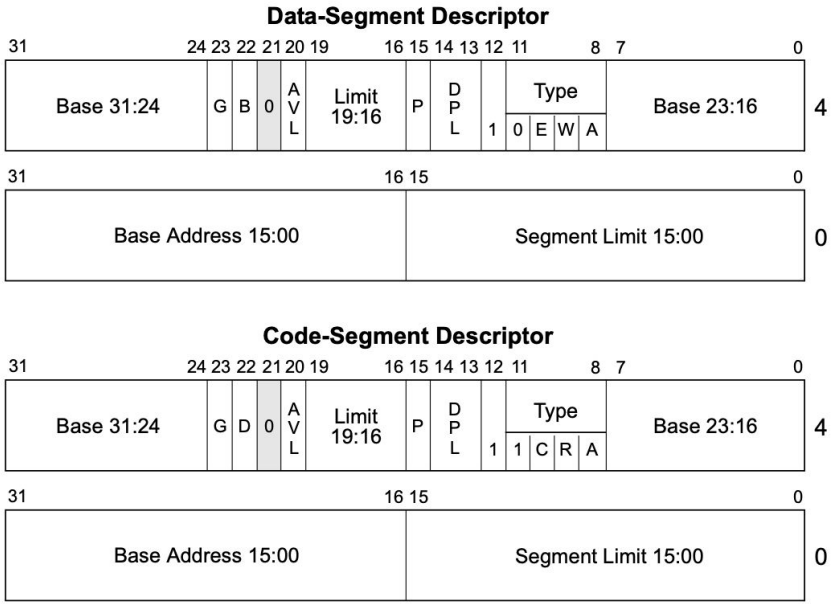
# Segmentación en x86

- 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)

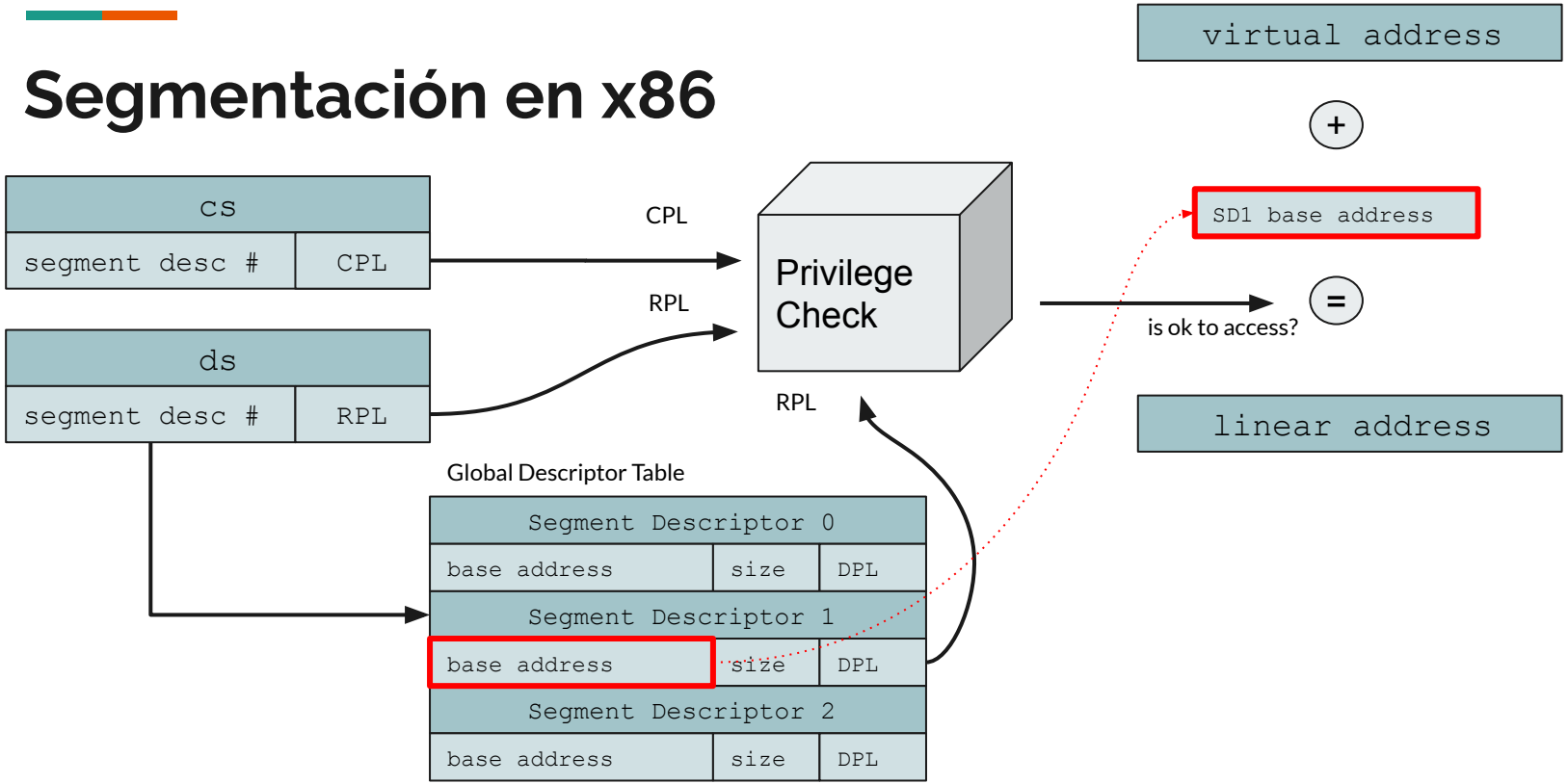


# Ejemplo de descriptor

- 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

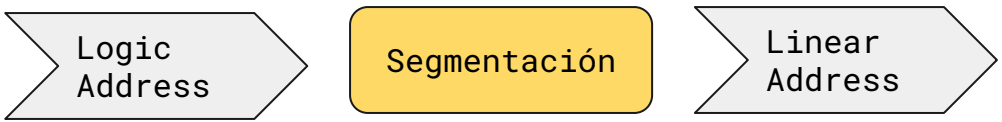


# Segmentación en x86



# 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



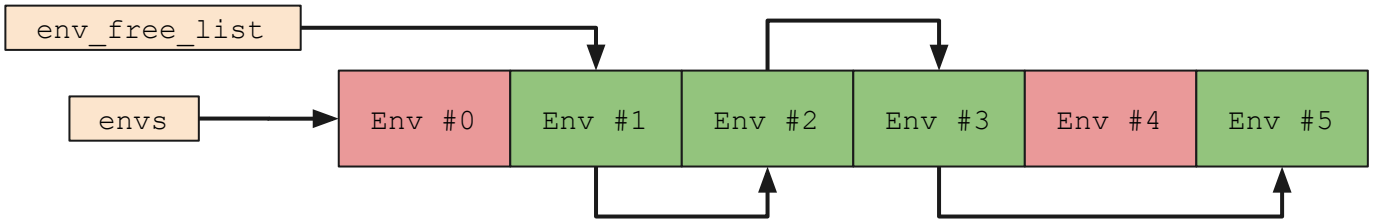
# Tarea 1 - El arreglo *envs*

# Tarea 1 - Inicializaciones

- **mem_init_envs()**
  - Reservar memoria para el arreglo *envs*... ¿Cómo pedimos la memoria?
- **env_init()**
  - Similar a **mem_init()**. Inicialización del arreglo *envs*... ¿Hay algún proceso corriendo?
- **env_alloc()** [*ya implementada*]
  - Estudiar cómo es el mecanismo de asignación de IDs
  - ¿Hay una relación entre el ID del proceso y su ubicación en el array *envs*?

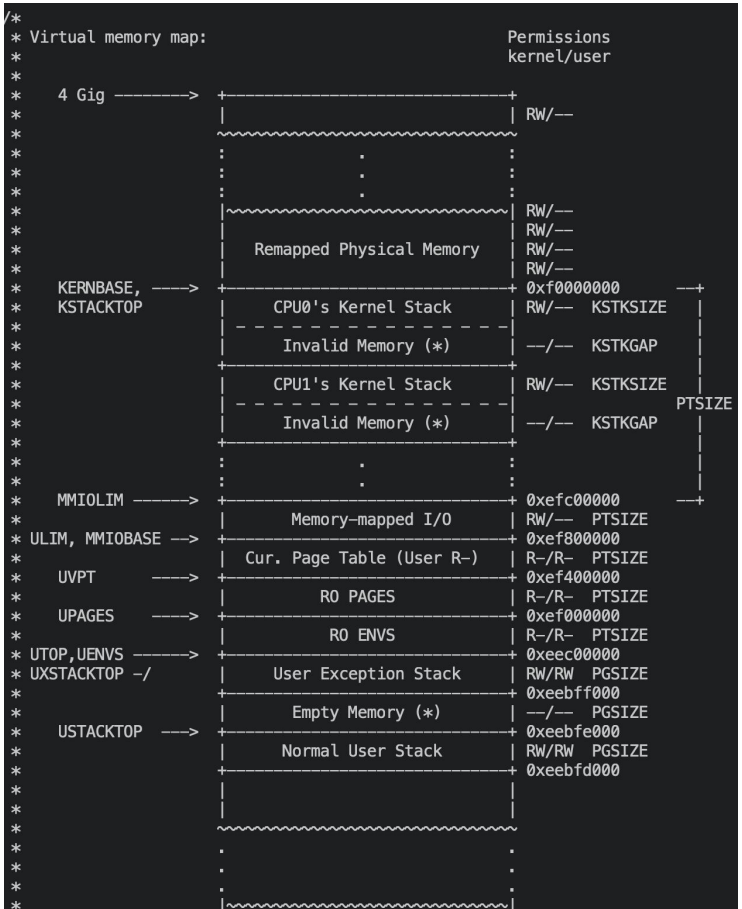
# Tarea 1 - Inicializaciones

- **mem_init_envs()**
  - Reservar memoria para el arreglo *envs*: usando **boot_alloc**
- **env_init()**
  - Similar a **mem_init()**. Inicialización del arreglo *envs*
  - Todos los procesos están libres: el **kernel** no es un proceso
- **env_alloc()** [*ya implementada*]
  - Estudiar cómo es el mecanismo de asignación de IDs
  - ¿Hay una relación entre el ID del proceso y su ubicación en el array *envs*?



# Tarea 1 - env_setup_vm()

- Configura el ***page directory*** del environment
- Se usa una **copia** del page directory del **kernel**
  - ¿Por qué? ¿Funciona?
  - ¿Qué cosas ya vienen mapeadas?
- En esta etapa **no se modifica cr3**
  - Porque si no, pasaremos a usar el *page directory* del proceso y no el del kernel
- ¿Qué nos falta?



# Tarea 2 - Carga de binarios

# Nos falta el código!

- Hay que armar el espacio de direcciones del proceso con cada uno de los segmentos
- Necesitamos **un binario** de donde extraer cada sección y...
- Un **formato** o **convención** para saber cómo mapearlo.

***Heap***: No tenemos, hasta que se implemente algún equivalente a s*brk()*

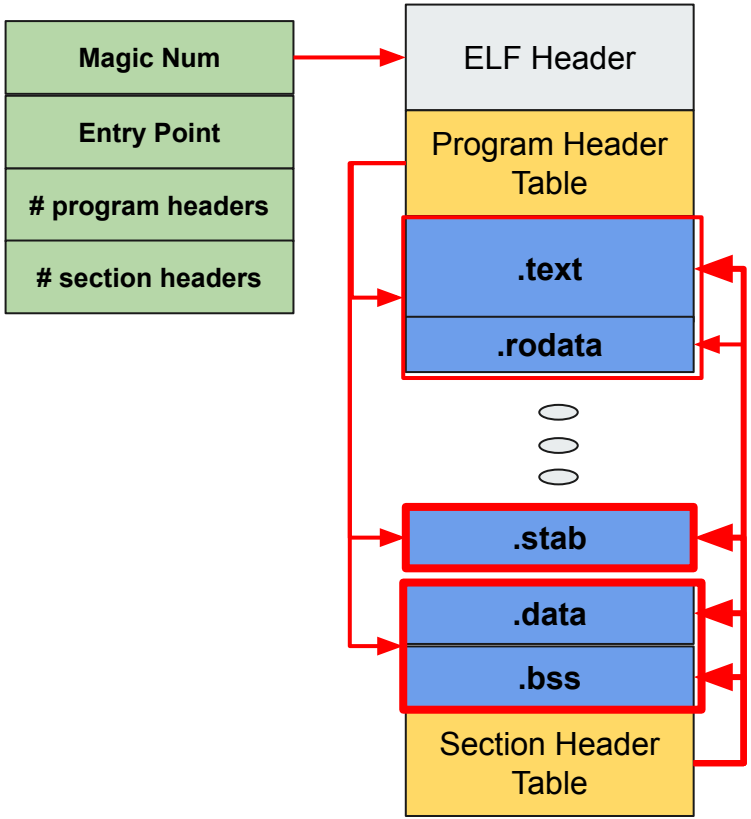
***Data***: Similar a Code, ¿de dónde sale?

***Stack***: ¿Tenemos stack? Hay reservado un espacio en ***USTACK*** para mapearlo, pero ¿Está mapeado ya?

***Code***: ¿Dónde mapeamos el código? ¿De dónde lo sacamos?

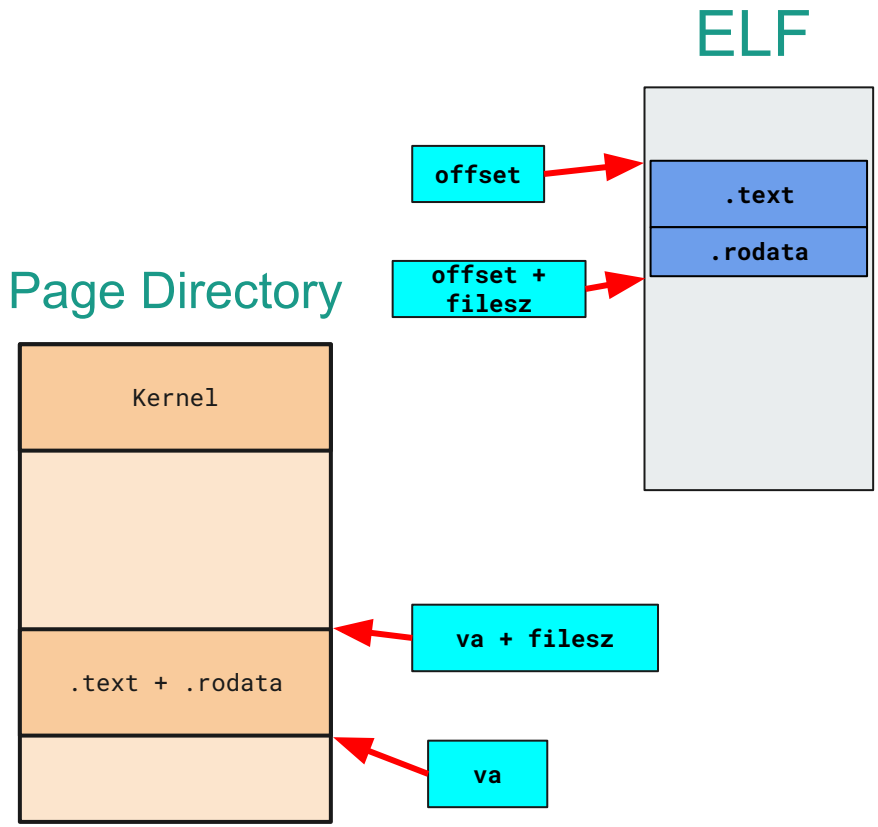
# Formato ELF

- 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



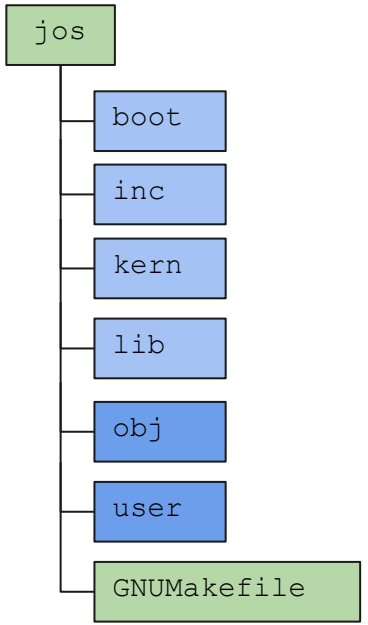
# Program Headers

- Indican un **mapeo en memoria** de una **sección** del binario
  - Indican una **dirección virtual**
  - Un **offset** dentro del binario
  - Un tamaño del bloque a copiar (**filesz**)
  - Y un tamaño de la región de memoria resultante (**memsz**)
- ¿Puede ser **memsz** más grande que **filesz**? ¿Y al revés?



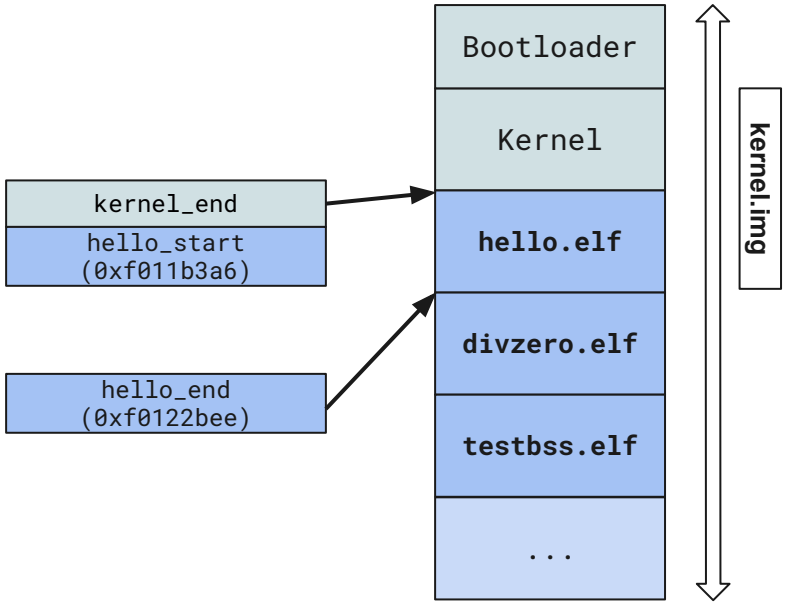
# Los programas de usuario

- 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?



# 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()**



# Tarea 2 - Carga del binario

- **region_alloc()**
  - Función auxiliar para reservar una región de memoria contigua con sucesivas llamadas a **page_insert()**
- **load_icode()**
  - Carga el binario a partir de un puntero a **struct Elf** (ver documentación)
  - Alocar memoria para cada program header de tipo LOAD y **copiar** desde el binario tantos bytes como el header use
  - También configura stack y entrypoint
  - ¿Tenemos que tener algún cuidado al llamar a **memcpy** o **memset**? ¿Por qué?
- **env_create()**
  - Combina todo lo visto hasta ahora para crear un proceso nuevo y cargarle el código

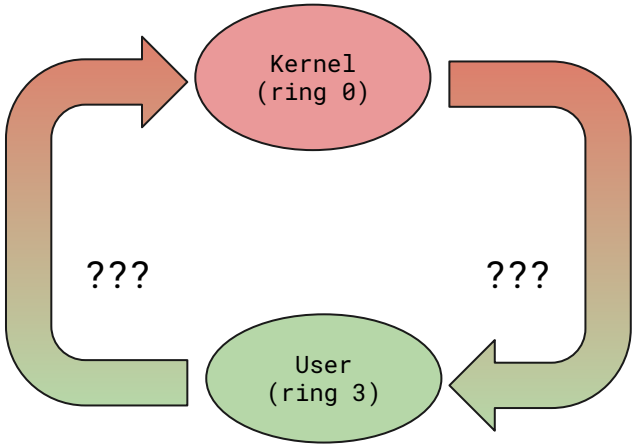
# Tarea 3 - Ejecución de un proceso

# ¿Cómo ejecutamos un proceso?

- Teniendo el **struct Env** listo… ¿cómo lo ejecutamos?
  - ¿Qué quiere decir “ejecutar” un proceso?
- ¿Cómo pasamos al ring 3?
- ¿Qué pasa si hay un programa malicioso?
- ¿Qué pasa si hay un programa con un loop infinito?
- ¿Cómo evitamos accesos indebidos al código del kernel?
- ¿Cómo permitimos que se vuelva al ring 0 para ejecutar una syscall?

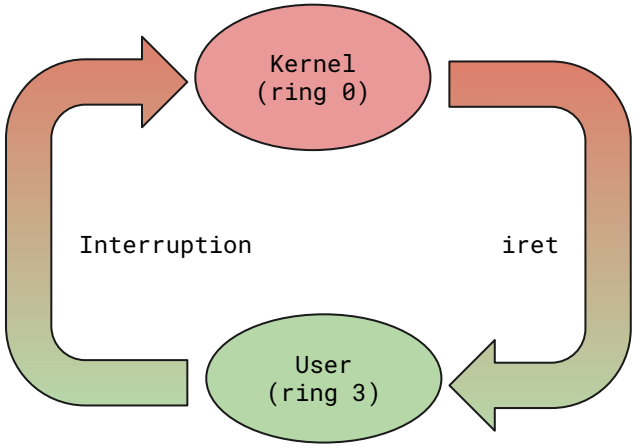
# Cambio de contexto

- 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?



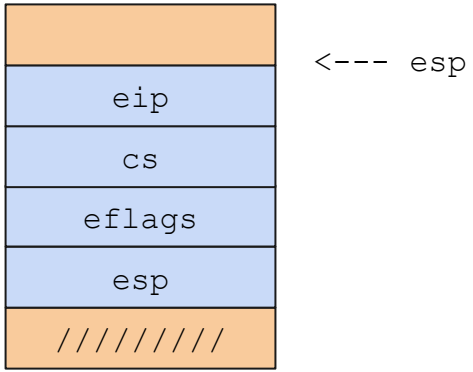
# Cambio de contexto

- 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**



# 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?



# 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?

# Tarea 3 - Ejecución de proceso

- **env_pop_tf()** [*en assembly*]
  - Esta es la última función que ejecuta el kernel antes de correr un proceso
  - No retorna!! ¿Por qué?
  - ¿Qué se tiene que cumplir para que el cambio de contexto funcione?
- **env_run()**
  - Cambia el estado del proceso que va a correr a **ENV_RUNNING**
  - Configura el *page directory* y llama a **env_pop_tf**
- gdb_hello
  - Seguimiento de la corrida de un programa
  - ¿Qué pasa luego de iret?
  - ¿Qué pasa cuando el usuario hace una syscall?