TP Scheduling

Sistemas Operativos - FIUBA

Integración de esqueleto

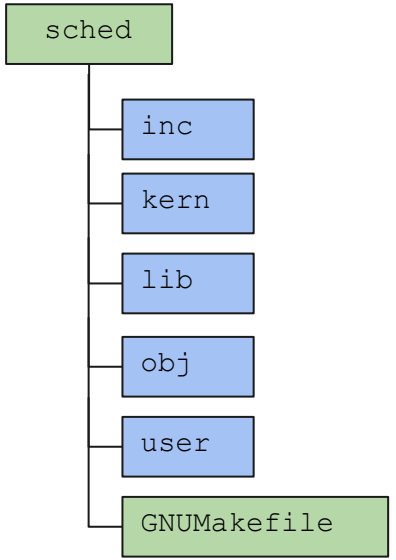
TP Sched - Introducción

- El esqueleto para este TP está en la rama *main* del repo **sched-skel**
- Para integrarlo deben:
  - desde su branch *entrega_malloc* crear una nueva rama **base_sched**
  - mergear los cambios de tp sched
  - pushear base_sched y crear entrega_sched

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

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

// Agregar el repositorio del esqueleto como remoto
$ git remote add sched
git@github.com:fisop/sched-skel.git

// Integración del esqueleto del tp sched
$ git fetch --all
$ git merge sched/main

// Pushear la rama base_sched
$ git push origin base_sched

// Creación de la rama de entrega
$ git checkout -b entrega_sched
$ git push -u origin entrega_sched
```

TP Sched - Navegando el repositorio

- Código en **C** con algunas secciones en **asm**
- El repo está dividido en varias carpetas
  - **inc**: encabezados comunes (**IMPORTANTE**!)
  - **kern**: el código del kernel!
  - **lib**: código de librerías de usuario
  - **user**: código de programas de usuario
  - **obj**: directorio de los binarios compilados



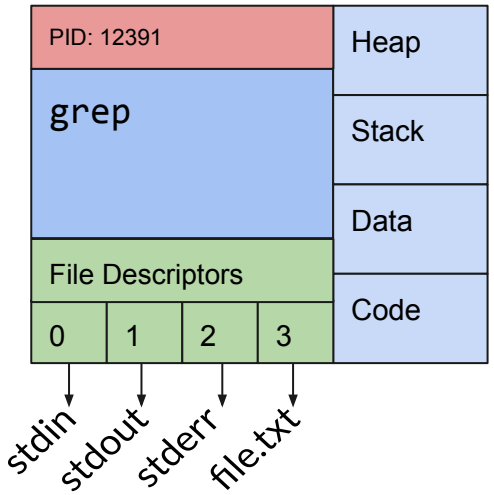
TP Sched - Corriendo JOS

- **make qemu**: corre JOS con modo gráfico
  - IMPORTANTE: **C-a x** para salir!
- **make qemu-nox**: corre JOS sin modo gráfico (redirige salida a la terminal)
- **make grade**: corre las pruebas automatizadas
- **make qemu-nox-gdb**: inicializa qemu pero espera una conexión de gdb para debuggear
- **make gdb**: inicia una sesión de gdb para debuggear JOS

Introducción a procesos (en JOS)

Procesos en el JOS

- Los procesos en JOS se llaman **environments**
- Implementación
  - Cambios de contexto
  - Scheduler *round robin*
  - Scheduler avanzado con prioridades
- Ya implementado:
  - Subsistema de *memoria*
  - Manejo de PCB
  - Syscalls varias (incluyendo fork)



Procesos en el JOS

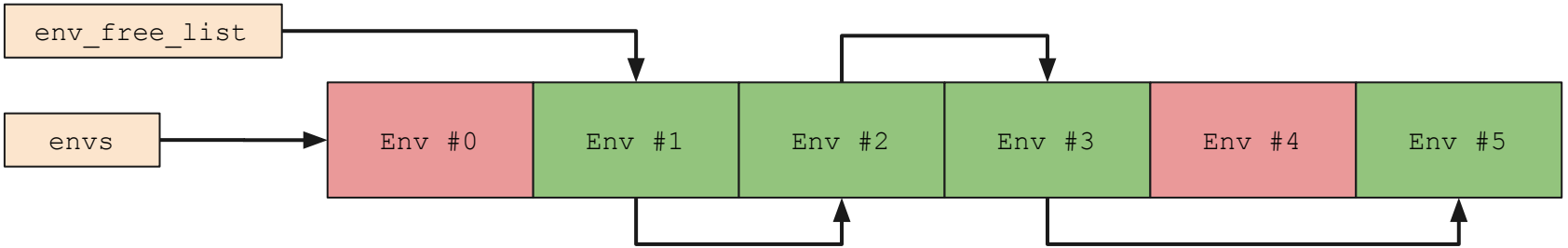
- Modelados con el *struct Env*
- Toda la información de un proceso
  - ID (*env_id*)
  - Estado del CPU (*env_tf*)
  - Memoria virtual (*env_pgdir*)
  - Estado
- Cada proceso tiene su propio ***page directory***

```
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

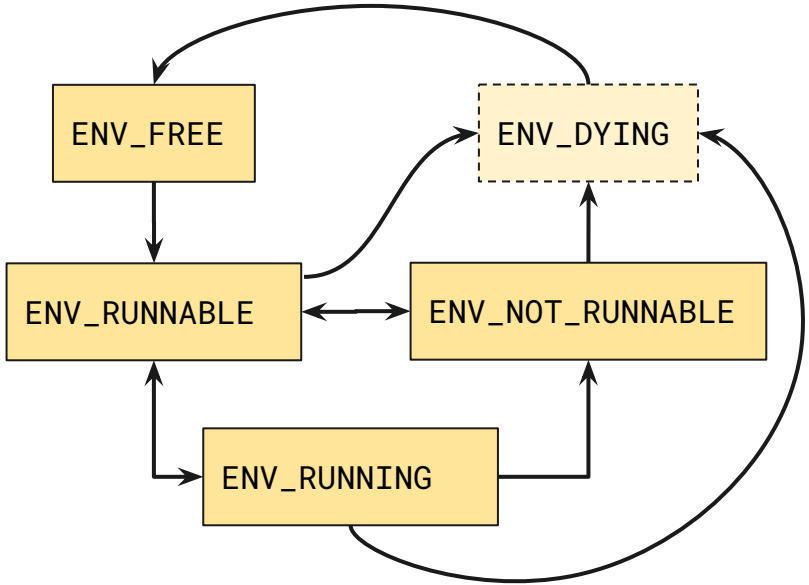
- El arreglo *envs* contiene **todos los procesos**
  - Similar a *pages*
  - Lista enlazada de “procesos libres”
- Variable *curenv* contiene al proceso actual

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



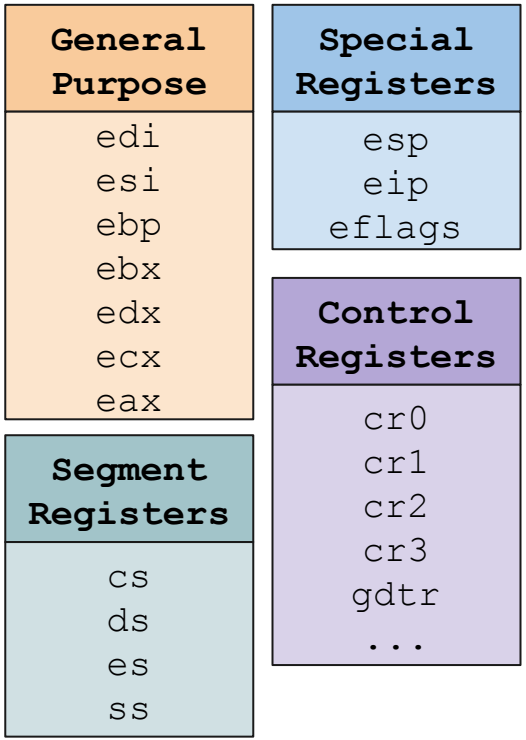
Procesos en JOS - Estados

- 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*



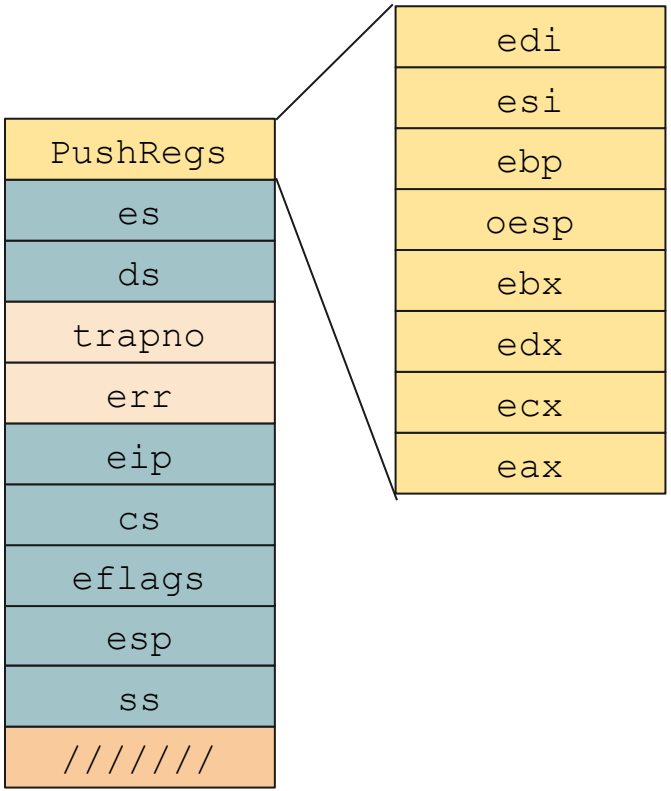
Los registros en x86

- 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**



El Struct Trapframe

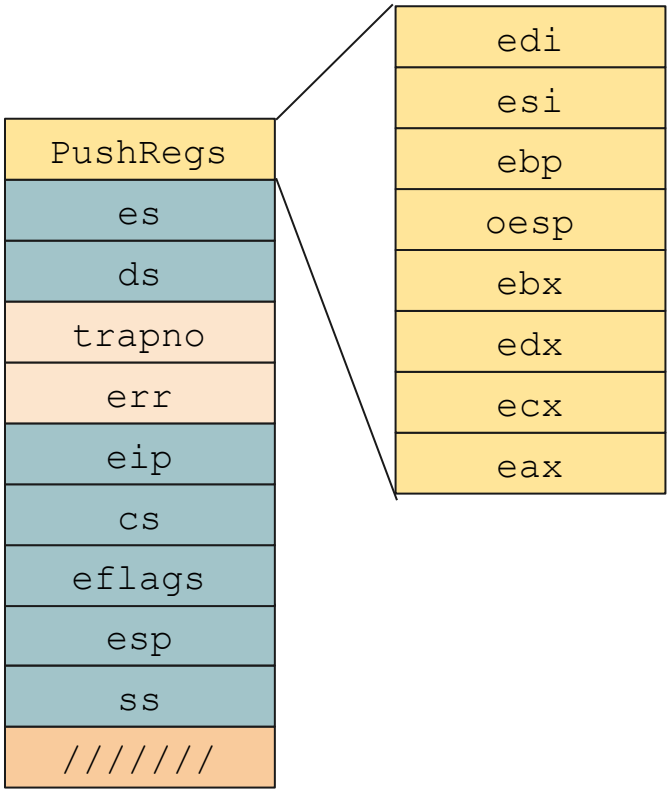
- Representa el estado de **los registros** de un environment determinado
- *PushRegs* son los registros de propósito general
  - están en el orden indicado para poder ser agregados o quitados al stack con **popal** y **pushal**
- ¿Hay un orden especial en los demás registros?



El Struct Trapframe - cont

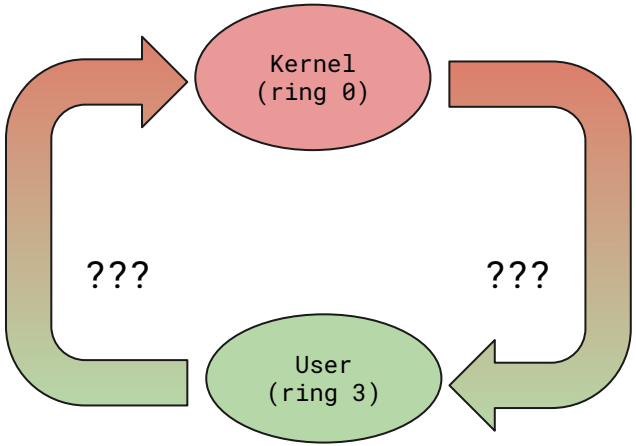
- **trapno** y **err** no son registros, si no campos auxiliares en JOS para indicar el número de *interrupción* y la presencia de error

- El resto de los registros componen la parte más importante del **contexto:**
  - **eip** nos dice *donde* estamos ejecutando código
  - **cs** codifica en sus bits de privilegio el **ring**
  - **esp** marca dónde está el stack (necesario para algunas instrucciones)
  - **eflags** contiene bits de estado incluyendo algunas configuraciones



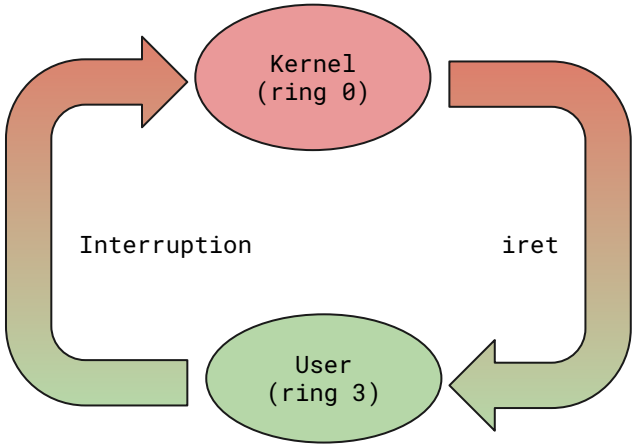
Cambio de contexto

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



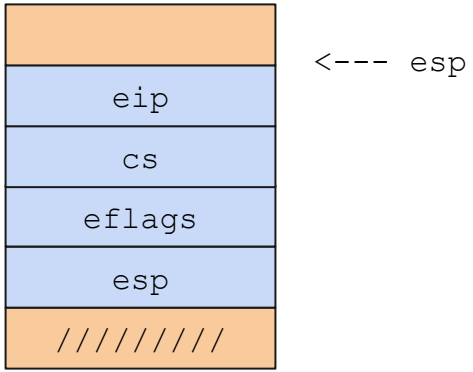
Cambio de contexto

- 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**



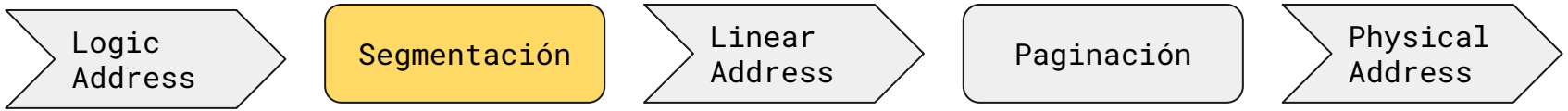
Pasando a modo usuario - iret

- 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?



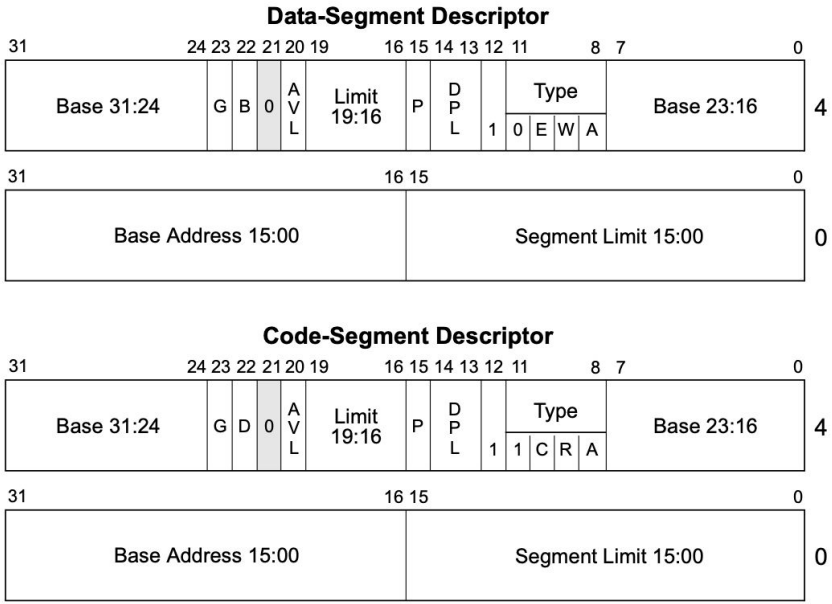
Segmentación en x86

- 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)

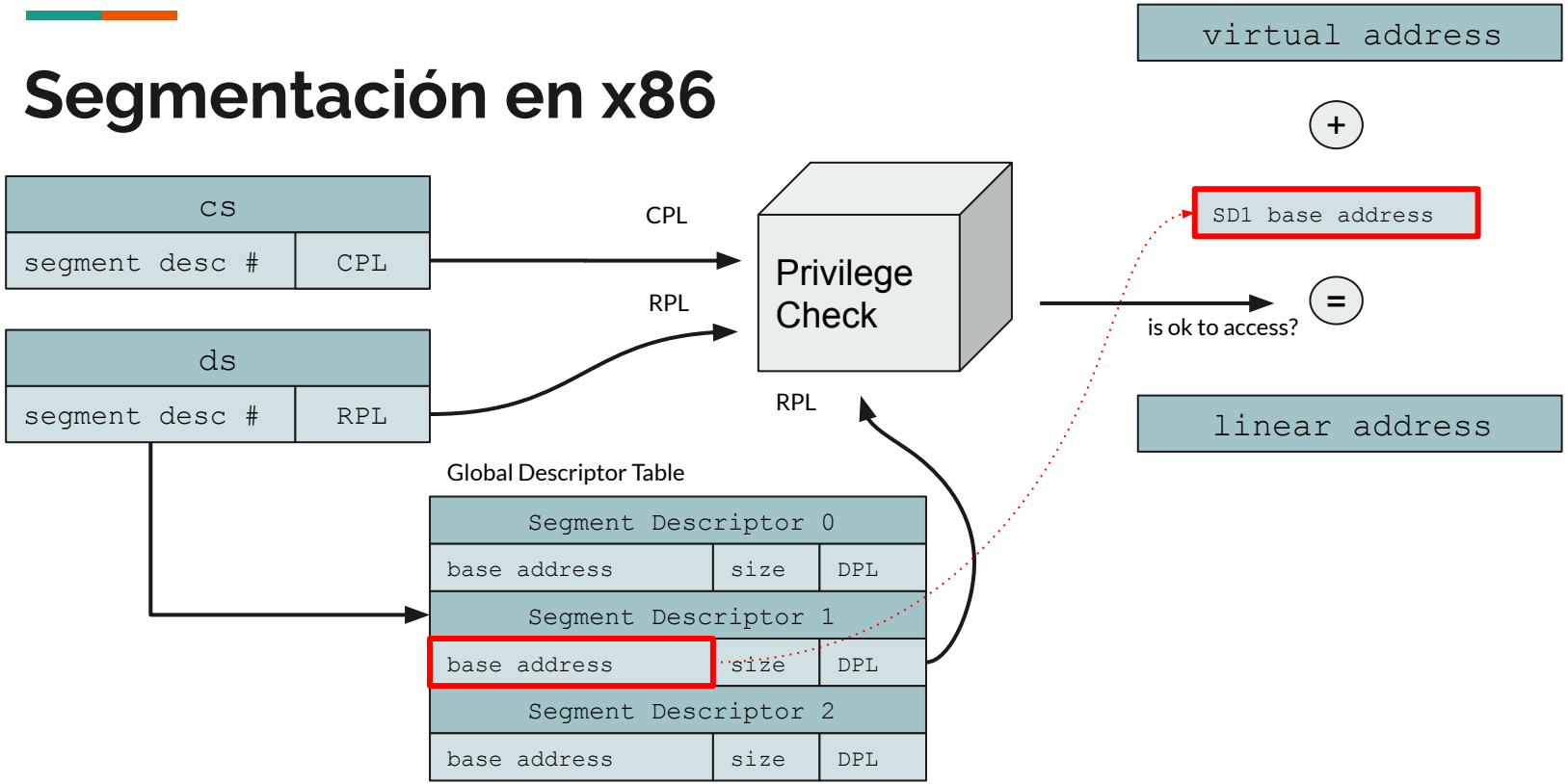


Ejemplo de descriptor

- 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