Inciso 2

Al contar con un sistema en modo protegido con segmentación flat y paginación activa podemos suponer que existen 4 entradas en la GDT que abarcan toda la memoria

- lectura/escritura de datos con dpl 0
- lectura/escritura de datos con dpl 3
- ejecución de código con dpl 0
- ejecución de código con dpl 3

Voy a asumir que ocupan las posiciones 10, 11, 12 y 13 de la GDT. También voy a asumir que en las posiciones 14, 15, 16 y 17 se encuentran los descriptores de las tss de las tareas A, B, C y D. Estas tss van a ser estructuras de tipo tss_t (definida más abajo) dentro de un arreglo llamado TSSs, donde sus atributos más relevantes son:

- eip = 0x05000000
- esp = 0x0540FA00
- cr3 = un puntero a un directorio de páginas que sea coherente con lo definido en el punto anterior
- ss0 = un selector de segmento de código de nivel 0
- cs = un selector de segmento de código de nivel 3
- resto de los registros de segmento = selector de segmento de datos de nivel 3
- esp0 = el final de una página del área libre del kernel

También es necesaria en la GDT una entrada que apunte a un TSS inicial. Este puede estar lleno de ceros ya que su contenido no importa, solo es necesario para hacer el primer task switch.
Dicho esto, procedo a mostrar el código. Más abajo irán apareciendo funciones y estructuras auxiliares a medida que aparezcan.

```
; ; PIC
extern pic_finish1
; ; Sched
extern sched_next_task
sched_task_offset:     dd 0xFFFFFFFF
sched_task_selector:   dw 0xFFFF
global _isr32
_isr32:
        pushad
        call pic_finish1 ; Avisar al pic que se recibio la interrupcion
        call sched_next_task
        mov word [sched_task_selector], ax ; se carga el selector de segmento
        jmp far [sched_task_offset]
        .fin:
        popad
        iret
```

La función sched_next_task devuelve el selector de segmento de la próxima tarea a ejecutar. Lo guardo en una variable global y luego hago un jump far donde el selector es el devuelto por la función y el offset es un número arbitrario. Este no es relevante ya que sólo con el selector de la tss deseada es suficiente para que se ejecute un task switch.

```
static __inline __attribute__((always_inline)) void outb(uint32_t port,
                                                         uint8_t data) {
  __asm __volatile("outb %0,%w1" : : "a"(data), "d"(port));
}
void pic_finish1(void) { outb(0x20, 0x20); }
```

```
#define GDT_RPL_RING_0 0

typedef struct task_struct {
  uint8_t idx_gdt;
  uint32_t data_limit;
} task_t;

task_t tasks[4] = {
  [0] = {
    .idx_gdt = (14 << 3) | GDT_RPL_RING_0,
    .data_limit = 0x5000000 + 0xFA00 - 1,
  },
  [1] = {
    .idx_gdt = (15 << 3) | GDT_RPL_RING_0,
    .data_limit = 0x5000000 + 0xFA00 - 1,
  },
  [2] = {
    .idx_gdt = (16 << 3) | GDT_RPL_RING_0,
    .data_limit = 0x5000000 + 0xFA00 - 1,
  },
  [3] = {
    .idx_gdt = (17 << 3) | GDT_RPL_RING_0,
    .data_limit = 0x5000000 + 0xFA00 - 1,
  },
};

uint8_t curr_idx = 0;

uint16_t sched_next_task(void) {
  curr_idx = (curr_idx + 1) % 4;
  return tasks[curr_idx].idx_gdt;
}
```

La función sched_next_task va rotando por las 4 tareas y devuelve el selector de segmento de la tss correspondiente.
También es necesario que en la IDT haya una entrada en el índice 32 cuyo offset apunte al handler definido anteriormente, su selector de segmento sea uno de código de nivel 0 con RPL 0 y que sus atributos son los correspondientes a un descriptor de interrupciones de 32 bits con DPL de supervisor (0x8E00)

Inciso 3

El servicio SetDataLimit toma en EAX tomar la cantidad total de paginas que debe tener el área de datos de la tarea actual y no retorna ningún dato. Ahora paso a explicar como funcionará.
Recordemos que en la estructura task_t tenemos un campo llamado data_limit, que guarda el último byte de offset dentro del área de datos de la tarea al que se puede acceder: cualquier dirección posterior al límite no está mappeada. Por lo tanto, si el usuario pide más páginas de las que tiene mapeadas actualmente, se llamará a kMallocPage() hasta llegar a la cantidad deseada, mientras que voy mapeando las direcciones físicas que devuelve la función. Si el usuario quiere achicar su área de datos, se irán desmapeando las páginas desde el límite actual hasta la cantidad de páginas deseadas, mientras que se va llamando a kFreePage() para liberar la página.

El servicio ReadData toma por EAX el valor ASCII de A, B, C o D mayúsculas y en EBX offset dentro del área de datos de la tarea de byte que se busca leer. En ECX se debe pasar un puntero en donde se guardará el byte leído y en EDX debe pasar un puntero en donde se guardará 1 o 0, dependiendo de si fue exitosa la lectura o no.
Para lograr esto, deberemos checkear si el caracter es válido y si el offset está dentro del límite de la tarea que se pretende leer. Si todo esto es correcto, se mapeara temporalmente la página correspondiente, se leerá el byte y se desmapeará la página.

Inciso 4

Sería necesario que en la IDT hayan entradas para atender a estos dos servicios. Cualquier índice que no entre en conflicto con los definidos en la arquitectura pueden funcionar. Tomaremos los índices 88 y 89. En esas entradas de la IDT, el campo offset apuntará a los handler definidos como _isr88 y _isr89. El campo segsel tendrá el valor de un selector de código de nivel 0 con RPL 0 y el campo de atributos corresponde a un descriptor de interrupciones de 32 bits con DPL de usuario (0xEE00) para que las tareas puedan llamar a los servicios.

```
extern SetDataLimit
extern ReadData
_isr88: pushad
        push eax
        call SetDataLimit
        add esp, 4
        popad
        iret
_isr89: pushad
        push edx
        push ecx
        push ebx
        push eax
        call ReadData
        add esp, 16
        popad
        iret
```

```
void SetDataLimit(uint32_t cantPaginas) {
  uint32_t cantPaginasActuales = tasks[curr_idx].data_limit / PAGE_SIZE;

  if (cantPaginas > cantPaginasActuales) {

    for (uint32_t i = cantPaginasActuales + 1; i <= cantPaginas; i++) {
      uint32_t physical = kMallocPage();
      mmu_map_page(TSSs[curr_idx].cr3, VIRT_START + CODE_SIZE + i * PAGE_SIZE,
physical, 1, 1);
    }

  } else if (cantPaginas < cantPaginasActuales) {

    for (uint32_t i = cantPaginasActuales; i > cantPaginas; i--) {
      uint32_t virtual = VIRT_START + CODE_SIZE + i * PAGE_SIZE;
      uint32_t phy_page_start = getPhyStart(curr_idx, virtual);
      kFreePage((virtual_addr & 0XFFF) + phy_page_start);//inicio de la página física
+ offset dentro de la página
      mmu_unmap_page(TSSs[curr_idx].cr3, VIRT_START + CODE_SIZE + i * PAGE_SIZE);
    }
  }
  tasks[curr_idx].data_limit = cantPaginas * PAGE_SIZE - 1;
}
```

La función no checkea que la cantidad de páginas pedidas sea menor al máximo disponible: es precondición que quien llama al servicio lo haga con un parámetro válido.

```
void ReadData(char tarea, uint32_t offset, int8_t* res, int8_t* succeed) {
  int8_t task_index = tarea - 65; //codigo ascii de A
  if(task_index < 0 || task_index> 3 || offset > tasks[task_index].data_limit){
    *succeed = 0;
  } else {
    uint32_t virtual_addr = VIRT_START + CODE_SIZE + offset;
    uint32_t phy_page_start = getPhyStart(task_index, virtual_addr);
    int8_t* virt_page_start = (int8_t*) 0x400000;
    mmu_map_page(TSSs[curr_idx].cr3, virt_page_start, phy_page_start, 1, 1);
    uint32_t offset_in_page = virtual_addr & 0XFFF;//inicio de la página virtual +
offset dentro de la página
    *res = virt_page_start[offset_in_page];
    mmu_unmap_page(TSSs[curr_idx].cr3, virt_page_start);
    *succeed = 1;
  }
}
```

Mapeo temporalmente la página donde se encuentra el byte a leer en una dirección virtual donde no van a haber conflictos (el final de la memoria reservada para el kernel). Luego accedo al byte deseado tomando el inicio de la página mapeada y desplazándome con los últimos 12 bits de la hipotética dirección virtual de dicho byte.

```
uint32_t getPhyStart(uint8_t task_index, uint32_t virt) {
  str_page_directory_entry* cr3 = TSSs[task_index].cr3;

  uint32_t directoryIdx = virt >> 22;//calculo el indice dentro del PD
  uint32_t tableIdx = (virt >> 12) & 0X3FF;//calculo el indice dentro del PT

  page_directory_entry PDE = cr3[directoryIdx];
  page_table_entry* PT = (page_table_entry*) PDE.page_table_base << 12; //calculo el
puntero a la page table
  return PT[tableIdx].physical_address_base << 12
}
```

```
#define INICIO_DE_PAGINAS_LIBRES 0x100000
uint32_t proxima_pagina_libre = INICIO_DE_PAGINAS_LIBRES;
uint32_t mmu_next_free_kernel_page(void) {
  uint32_t pagina_libre = proxima_pagina_libre;
  proxima_pagina_libre += PAGE_SIZE;
  return pagina_libre;
}
void mmu_map_page(page_directory_entry* cr3, uint32_t virt, uint32_t phy, uint8_t us,
uint8_t rw) {
  uint32_t directoryIdx = virt >> 22;
  uint32_t tableIdx = (virt >> 12) & 0X3FF;

  if (cr3[directoryIdx].present == 0)   {
    uint32_t newPT = mmu_next_free_kernel_page();
    for (int i = 0; i < 1024; i++)     {
      ((uint32_t *)newPT)[i] = 0;
    }
    cr3[directoryIdx].page_table_base = newPT >> 12;
    cr3[directoryIdx].present = 1;
    cr3[directoryIdx].user_supervisor = us;
    cr3[directoryIdx].read_write = rw;
  }
  page_table_entry* PT = (page_table_entry*) cr3[directoryIdx].page_table_base << 12;
  PT[tableIdx].physical_address_base = phy >> 12;
  PT[tableIdx].present = 1;
  PT[tableIdx].user_supervisor = us;
  PT[tableIdx].read_write = rw;
  tlbflush(); // para invalidar la Translation Lookaside Buffer
}
```

Esta función se encarga de mappear una dirección virtual a una física en un directorio de páginas dado. En caso de ser necesarias nuevas Page Tables, estas se ubicaran en el espacio libre del kernel, que debe empezar en la dirección 0x100000.

```
void mmu_unmap_page(page_directory_entry *cr3, vaddr_t virt) {
  uint32_t directoryIdx = virt >> 22;
  uint32_t tableIdx = (virt >> 12) & 0X3FF;
  page_directory_entry PDE = cr3[directoryIdx];
  uint32_t PT = PDE.page_table_base << 12;
  ((page_table_entry *)PT)[tableIdx].present = 0;
  tlbflush();
}
```

Esta función pone el bit de presente en 0 en la PTE correspondiente a la dirección y cr3 pasados.

```
typedef struct str_page_table_entry
{
  uint8_t present : 1;
  uint8_t read_write : 1;
  uint8_t user_supervisor : 1;
  uint8_t page_write_through : 1;
  uint8_t page_cache_disable : 1;
  uint8_t accessed : 1;
  uint8_t dirty : 1;
  uint8_t x : 1;
  uint8_t global : 1;
  uint8_t available : 3;
  uint32_t physical_address_base : 20;
} __attribute__((__packed__)) page_table_entry;

typedef struct str_page_directory_entry
{
  uint8_t present : 1;
  uint8_t read_write : 1;
  uint8_t user_supervisor : 1;
  uint8_t page_write_through : 1;
  uint8_t page_cache_disable : 1;
  uint8_t accessed : 1;
  uint8_t x : 1;
  uint8_t page_size : 1;
  uint8_t ignored : 1;
  uint8_t available : 3;
  uint32_t page_table_base : 20;
} __attribute__((__packed__)) page_directory_entry;
```

```
typedef struct str_tss {
  uint16_t ptl;
  uint16_t unused0;
  uint32_t esp0;
  uint16_t ss0;
  uint16_t unused1;
  uint32_t esp1;
  uint16_t ss1;
  uint16_t unused2;
  uint32_t esp2;
  uint16_t ss2;
  uint16_t unused3;
  uint32_t cr3;
  uint32_t eip;
  uint32_t eflags;
  uint32_t eax;
  uint32_t ecx;
  uint32_t edx;
  uint32_t ebx;
  uint32_t esp;
  uint32_t ebp;
  uint32_t esi;
  uint32_t edi;
  uint16_t es;
  uint16_t unused4;
  uint16_t cs;
  uint16_t unused5;
  uint16_t ss;
  uint16_t unused6;
  uint16_t ds;
  uint16_t unused7;
  uint16_t fs;
  uint16_t unused8;
  uint16_t gs;
  uint16_t unused9;
  uint16_t ldt;
  uint16_t unused10;
  uint16_t dtrap;
  uint16_t iomap;
} __attribute__((__packed__, aligned(8))) tss_t;
```