CARTUCHO

Clase pre-parcial

AyOC 2c2025

ORGA GENESIS

32-BIT VIDEO ENTERTAINMENT SYSTEM

ÚNETE A LA REVOLUCIÓN DE 32 BITS

- Procesador Intel x86
- Brillantes Gráficos de 32-bits a Color
- Sonido Estéreo Digital
- Incluye 3675 Juegos Clásicos

ENUNCIADO

ENUNCIADO

Nos encargaron el desarrollo de un kernel para la nueva consola Orga Génesis que cuenta con un procesador x86. Para ahorrar trabajo vamos a tomar nuestro kernel y expandirlo para permitir la ejecución de juegos mediante cartuchos, que contendrán el código y los recursos gráficos (sprites de personajes, fondos).

Se incorpora entonces un lector de cartuchos que, entre otras cosas, copiará los gráficos del cartucho a un buffer de video de 4 kB en memoria. Cada vez que el buffer esté lleno y listo para su procesamiento, el lector se lo informará al kernel mediante una interrupción externa mapeada al IRQ 40 de nuestro x86.

Distintas tareas se ocupan de mostrar los gráficos en pantalla y de realizar otros pre/postprocesamientos de imagen on-the-fly.

ENUNCIADO

El sistema debe soportar que las tareas puedan acceder al buffer de video a través de los siguientes mecanismos:

- DMA (Direct Memory Access): se mapea la dirección virtual **`0xBABAB000`** del espacio de direcciones de la tarea directamente al buffer de video.
- Por copia: se realiza una copia del buffer en una página física específica y se mapea en una dirección virtual provista por la tarea. Cada tarea debe tener una copia única.

Dicho buffer se encuentra en la dirección física de memoria **`0xF151C000`** y solo debe ser modificable por el lector de cartuchos.

ENUNCIADO

Para solicitar acceso al buffer, las tareas deberán informar que desean acceder a él mediante una syscall **`opendevice`** habiendo configurado el tipo de acceso al buffer en la dirección virtual **`0xACCE5000`** (mapeada como r/w para la tarea). Allí almacenará una variable **`uint8_t acceso`** con posibles valores **`0`**, **`1`** y **`2`**. El valor **`0`** indica que la tarea no accede al buffer de video, **`1`** que accede mediante DMA y **`2`** que accede por copia. De acceder por copia, la dirección virtual donde realizar la copia estará dada por el valor del registro **`ECX`** al momento de llamar a **`opendevice`** y sus permisos van a ser de r/w. Asumimos que las tareas tienen esa dirección virtual mapeada a alguna dirección física.

El sistema no debe retomar la ejecución de estas tareas hasta que se detecte que el buffer está listo y se haya realizado el mapeo DMA o la copia correspondiente.

Una vez que la tarea termine de utilizar el buffer, deberá indicarlo mediante la syscall **`closedevice`**. En ésta se debe retirar el acceso al buffer por DMA o dejar de actualizar la copia, según corresponda.

La *interrupción de buffer completo* será la encargada de dar el acceso correspondiente a las tareas que lo hayan solicitado y actualizar las copia del buffer "vivas". Es deseable que cada tarea que accede por copia mantenga una única copia del buffer para no ocupar la memoria innecesariamente.

Como las direcciones que utilizamos viven por fuera de los 817MB definidos en los segmentos, asumimos que los segmentos de código y datos de nivel 0 y 3 ocupan toda la memoria (4 GB)

ENUNCIADO

1. Syscalls
   a. Programar la rutina que atenderá la interrupción que el lector de cartuchos generará al terminar de llenar el buffer.
   b. Programar las syscalls **`opendevice`** y **`closedevice`**.

Cuentan con las siguientes funciones ya implementadas:

- **`void buffer_dma(pd_entry_t* pd)`** que dado el page directory de una tarea realice el mapeo del buffer en modo DMA.
- **`void buffer_copy(pd_entry_t* pd, paddr_t phys, vaddr_t virt)`** que dado el page directory de una tarea realice la copia del buffer a la dirección física pasada por parámetro y realice el mapeo a la dirección virtual pasada por parámetro.

2. Auxiliares
   a. Programar la función void buffer_dma(pd_entry_t* pd)
   b. Programar la función void buffer_copy(pd_entry_t* pd, paddr_t phys, vaddr_t virt)

LECTOR DE CARTUCHOS

- Pieza de hardware que copia los gráficos del cartucho a un buffer de video de 4 KB en memoria en la dirección física **`0xF151C000`**.
- Informará que el buffer está lleno al kernel mediante una **interrupción** externa mapeada al **IRQ 40** de nuestro x86.

TIPOS DE ACCESO AL BUFFER

- ***DMA***: se mapea la dirección virtual **`0xBABAB000`** del espacio de direcciones de la tarea directamente al buffer de video.
- ***Por copia***: se realiza una copia del buffer en una **página física específica** y se mapea en una **dirección virtual provista por la tarea** (cada tarea debe tener una copia única).

TAREAS - MODO DE ACCESO

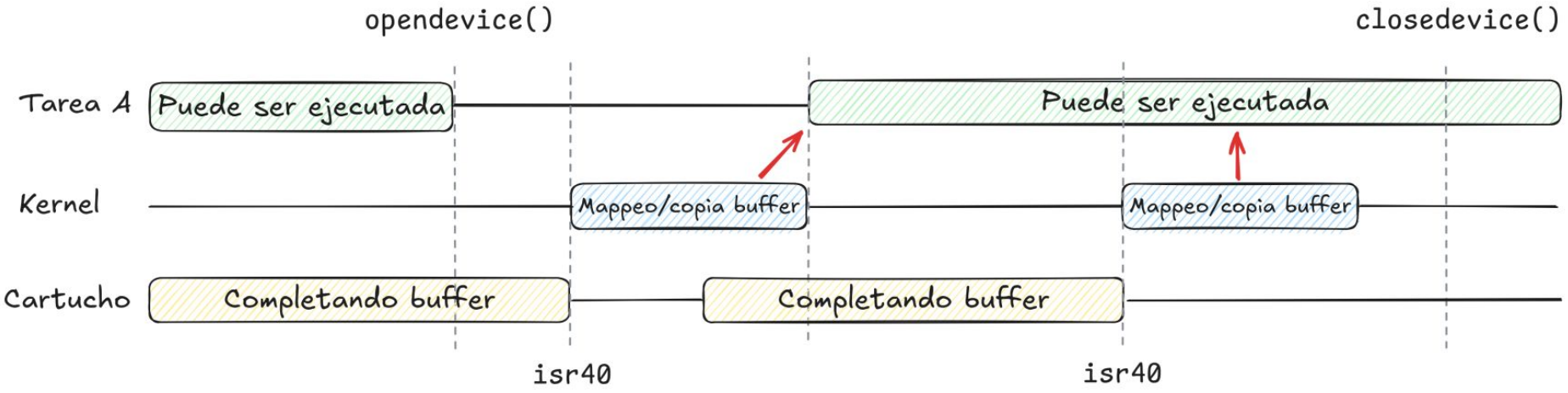
Cada tarea configura su modo de acceso en la dirección virtual **`0xACCE5000`** (mapeada como r/w para la tarea):

- **`0`**: no utiliza el buffer de video.
- **`1`**: accede mediante DMA.
- **`2`**: accede por copia.

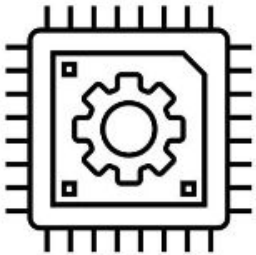
El acceso al buffer se controla a través de dos syscalls: **`opendevice`** y **`closedevice`**.

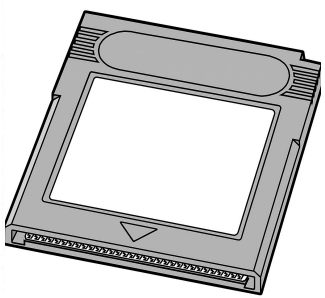
BUFFER COMPLETO

La interrupción de buffer completo (IRQ 40) será la encargada de dar el acceso correspondiente a las tareas que lo hayan solicitado y actualizar las copias del buffer “vivas”. Es deseable que cada tarea que accede por copia mantenga una única copia del buffer para no ocupar la memoria innecesariamente.

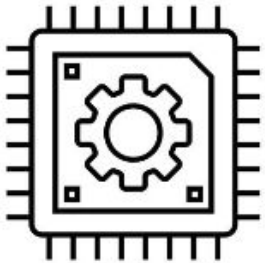


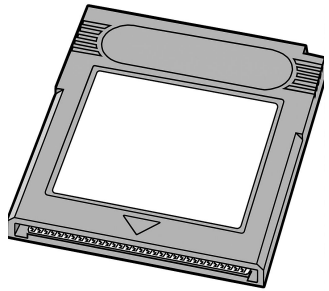
EJEMPLO



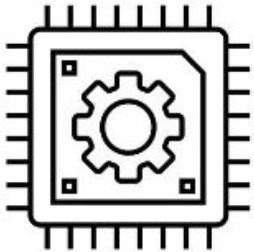


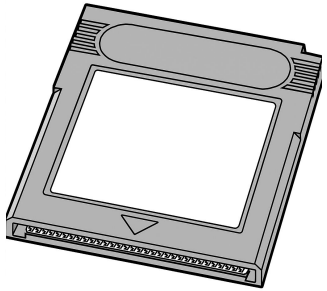
Buffer de memoria





Buffer de memoria



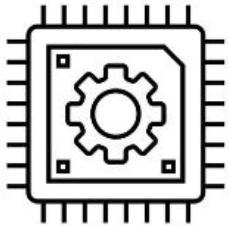


Tarea en ejecucion: 1

Nivel: Usuario

Cola de ejecucion: [R, W, W]

Buffer de memoria

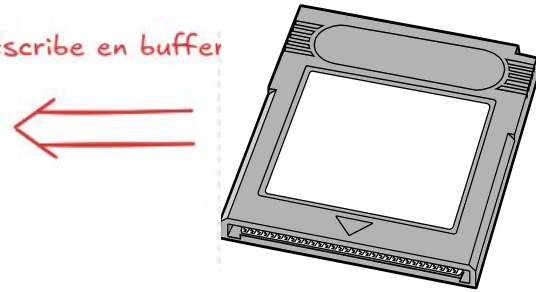


Tarea en ejecucion: 1

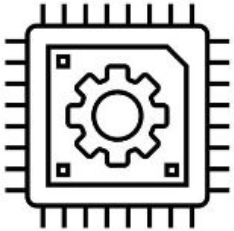
Nivel: Usuario

Cola de ejecucion: [R, W, W]

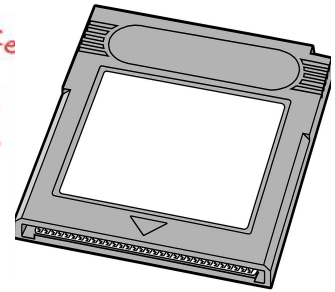
Escribe en buffer



Buffer de memoria



Escribe en buffe

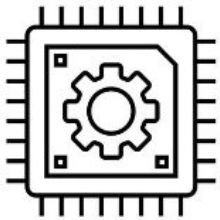


Tarea en ejecucion: 2

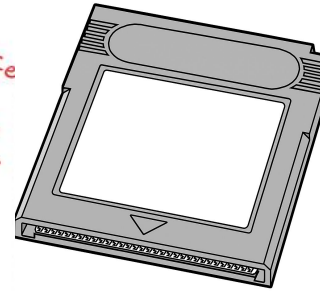
Nivel: Usuario

Cola de ejecucion: [W, R, W]

Buffer de memoria



Escribe en buffe

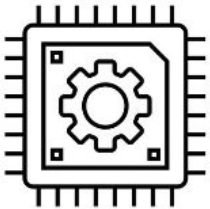


Tarea en ejecucion: 3

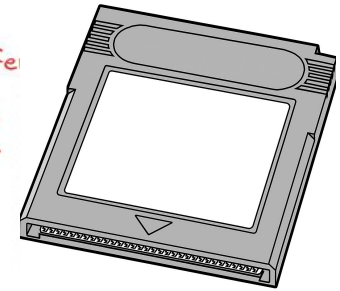
Nivel: Usuario

Cola de ejecucion: [W, B, R]

Buffer de memoria



Escribe en buffe

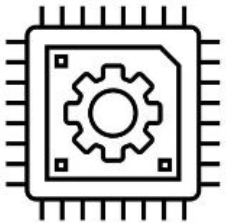


Tarea en ejecucion: 1

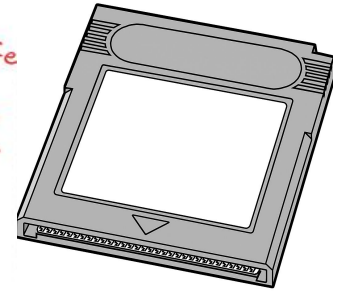
Nivel: Usuario

Cola de ejecucion: [R, B, W]

Buffer de memoria



Escribe en buffe

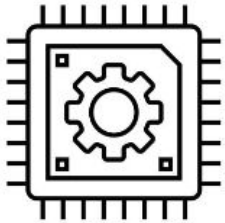


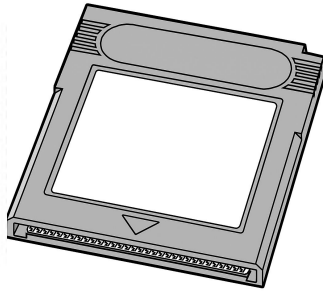
Tarea en ejecucion: 3

Nivel: Usuario

Cola de ejecucion: [B, B, R]

Buffer listo!



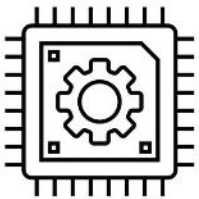


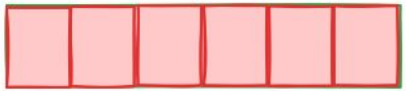
Tarea en ejecucion: 3

Nivel: Usuario

Cola de ejecucion: [B, B, R]

irq 40





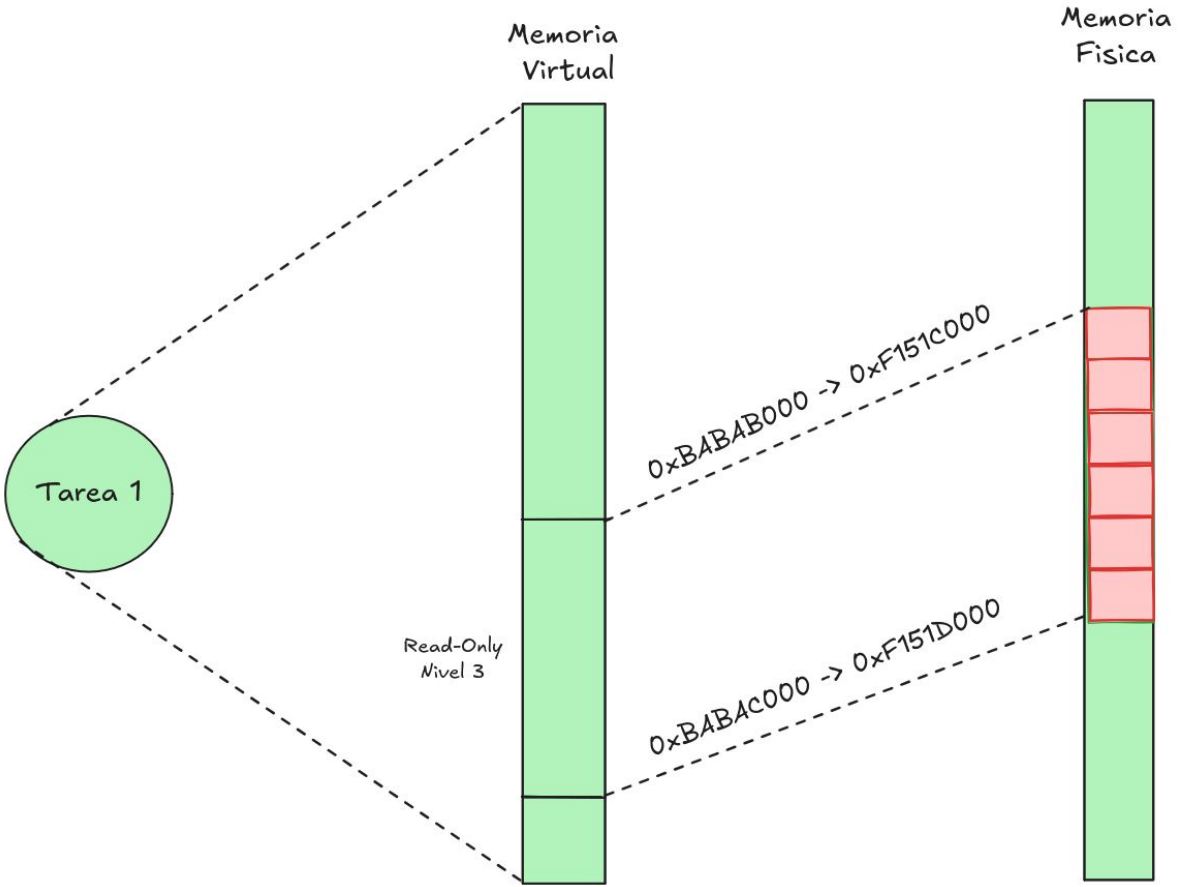
Una vez ocurrida la interrupcion,
el procesador debe realizar el mapeo del
buffer segun el modo de acceso

Tarea en ejecucion: 3

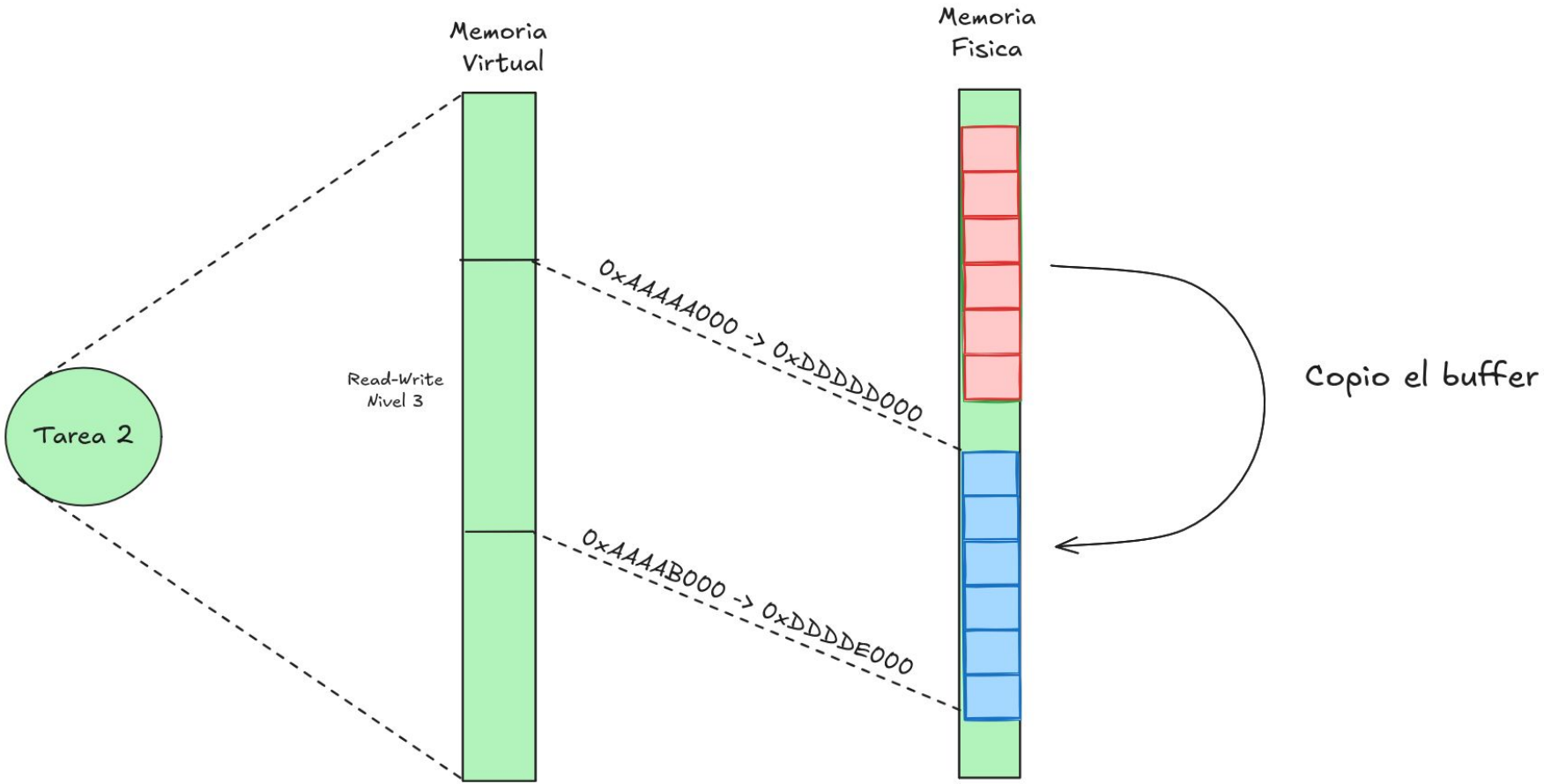
Nivel: Kernel

Cola de ejecucion: [B, B, R]

MAPEO POR DMA



MAPEO POR COPIA



RESOLVAMOS

PASOS A REALIZAR

Utilizaremos un enfoque top-down:

- Se nos pide resolver **`deviceready`**.
  - Encaremos una solución.
  - ¿Podemos resolverlo con el sistema que tenemos o es necesario agregar estructuras/funcionalidades nuevas?
- Implementar las syscalls de acceso al buffer:
  - **`opendevice`**: ¿Cómo podemos bloquear una tarea hasta el siguiente **`deviceready`**? ¿Qué pasa si una tarea solicita acceso más de una vez?
  - **`closedevice`**: ¿Cómo restauramos el estado de las tareas que pidieron acceso?.
- Implementar los mapeos de paginación (**`buffer_dma`** / **`buffer_copy`**).

ESTRUCTURAS

- `TASK_STATE`
  - Tenemos que agregarle información a la tarea sobre la dirección virtual que va a usar para mapear el buffer. En particular, podemos agregar un estado a una tarea la cual sea `BLOCKED`, la cual da un indicio de que esta bloqueada esperando a tener acceso al device.
- `IDT`
  - Tenemos que agregar una interrupción por hardware y dos syscalls.
    - Como la interrupción es de hardware, entonces el kernel es el único que puede atender a la misma.
    - Como las syscalls son servicios que provee el SO al usuario, entonces estas deben ser capaces de ser llamadas por un usuario, y el código que ejecutan es de nivel 0.
- `SCHED_ENTRY`
  - Esta estructura posee toda la informacion sobre la tarea.
  - Podemos agregarle la informacion sobre el modo de acceso que va a tener la tarea, y la direccion virtual en donde va a poder encontrar el buffer

ESTRUCTURAS

tasks.c

```
typedef struct {
  int16_t selector;
  task_state_t state;

  // Agregado para resolver
  // el mecanismo propuesto
  uint32_t copyDir;
  uint8_t mode;
} sched_entry_t;
```

idt.c

```
void idt_init() {
  // ...
  IDT_ENTRY0(40);
  // ...
  IDT_ENTRY3(90);
  IDT_ENTRY3(91);
  // ...
}
```

ESTRUCTURAS

tasks.c

```c
typedef enum {
  NO_ACCESS,
  ACCESS_DMA,
  ACCESS_COPIA
} task_access_mode_t;
```

```c
typedef enum {
  TASK_SLOT_FREE,
  TASK_RUNNABLE,
  TASK_PAUSED,
  // Nuevo estado de tarea
  TASK_BLOCKED
  // Estado para usar después
  TASK_KILLED
} task_state_t;
```

DEVICEREADY

Cuando se ejecuta esta función, entonces el kernel toma posesión de la tarea que estaba ejecutando, y empieza a mapear a todas las tareas que hayan solicitado acceso al área del buffer.

Para ello, la función debe:

- Iterar sobre **todas las tareas** definidas en el scheduler.
- Comprobar si **está esperando** para acceder o si **ya tiene acceso** al buffer.
- Actualizar las estructuras de paginación según corresponda:
  - La tarea está solicitando acceso:
    - Si es por DMA entonces tenemos que mapear la direccion virtual `0xBABAB000` a la fisica `0xF151C0000` con permisos de usuario y Read-Only
    - Si es por copia, mapeamos la dirección virtual pasada por parametro a una **nueva dirección física (En caso de primer mapeo)** y hacemos la copia de datos.
  - La tarea ya tiene acceso:
    - Si es por DMA no tenemos que hacer nada.
    - Si es por copia hay que actualizar la copia.

DEVICEREADY

Primero definimos el handler en assembler…

isr.asm

```
extern deviceready
global isr_40

isr_40:
    pushad
    call pic_finish
    call deviceready
    popad
    iret
```

DEVICEREADY

idt.c

```
void deviceready(void){
    for(int i = 0; i < MAX_TASKS; i++){
        sched_entry_t* tarea = &sched_tasks[i];
        if(tarea->mode == NO_ACCESS) // No solicita acceso al buffer
            continue;
        if(tarea->status == BLOCKED){
            if(tarea->mode == ACCESS_DMA)    // Solicita acceso en modo DMA
                buffer_dma(CR3_TO_PAGE_DIR(task_selector_to_CR3(tarea->selector)));

            if(tarea->mode == ACCESS_COPY)   // Solicita acceso en modo por copia
                buffer_copy(CR3_TO_PAGE_DIR(task_selector_to_CR3(tarea->selector)), mmu_next_user_page(), tarea->copyDir);

            tarea->status = TASK_RUNNABLE;   // Dejamos la tarea lista para correr en una próxima ejecución
        } else{
            if(tarea->mode == ACCESS_COPY){  // Actualizar copia
                paddr_t destino = virt_to_phy(task_selector_to_CR3(tarea->selector), tarea->copyDir);
                copy_page((paddr_t)0xF151C000, destino);
            }
        }
    }
}
```

OPENDEVICE

Syscall que permite a las tareas solicitar acceso al buffer según el tipo configurado. En el caso de acceso por copia, la dirección virtual donde realizar la copia estará dada por el valor del registro **`ECX`** al momento de llamarla.

El sistema **no debe retornar la ejecución** de las tareas que llaman a la syscall hasta que se detecte que el buffer está listo y se haya realizado el mapeo DMA o la copia correspondiente.

CLOSEDEVICE

Una vez la tarea termina de utilizar el buffer, debe indicarlo haciendo uso de esta syscall. En ella se debe retirar el acceso por DMA o dejar de actualizar la copia, según corresponda.

OPENDEVICE

isr.asm

```
extern opendevice
global isr_90
isr_90:
    pushad

    push ecx
    call opendevice
    add esp, 4
    call sched_next_task

    mov word [sched_task_selector], ax
    jmp far [sched_task_offset]

    popad
    iret
```

Aclaraciones:

- Al usar convención de 32 bits, los parámetros se pasan por pila.
- sched_next_task nos va a devolver en ax un selector de tarea distinto al que está ejecutando.
- Recordemos que si sched_next_task no encuentra un selector de una tarea de su lista, entonces devuelve el selector de la tarea idle.

OPENDEVICE

idt.c

```
uint8_t opendevice(uint32_t copyDir){
    sched_tasks[current_task].status = TASK_BLOCKED;
    sched_tasks[current_task].mode = *(uint8_t*)0xACCE5000;
    sched_tasks[current_task].copyDir = copyDir;
}
```

CLOSEDEVICE

isr.asm

```
extern closedevice
global isr_91
isr_91:
    pushad
    call closedevice
    popad
    iret
```

CLOSEDEVICE

idt.c

```
void closedevice(void){
    // En el caso DMA, la dir virtual de la pagina es siempre la misma
    if(sched_tasks[current_task].mode == ACCESS_DMA)
        mmu_unmap_page(rcr3(), (vaddr_t)0xBABAB000);

    // En el caso por copia, la dir virtual la tenemos en el struct del scheduler
    if(sched_tasks[current_task].mode == ACCESS_COPY)
        mmu_unmap_page(rcr3(), sched_tasks[current_task].copyDir);

    sched_tasks[current_task].mode = NO_ACCESS;
}
```

FUNCIONES DE MAPEO

Se nos pide implementar el mapeo asociado a los modos de acceso al buffer de video del cartucho. Recordemos:

- Si es por DMA entonces tenemos que mapear como solo lectura **`0xBABAB000`** (virtual) a **`0xF151C0000`** (física).
- Si es por copia, mapeamos la dirección virtual parámetro a la dirección física parámetro y luego hacemos la copia de la página que comienza en **`0xF151C000`**.

BUFFER(DMA/COPY)

mmu.c

```
void buffer_dma(pd_entry_t pd){
    mmu_map_page(   (uint32_t)pd, (vaddr_t)0xBABAB000,
                    (paddr_t)0xF151C000, MMU_U | MMU_P);
}

void buffer_copy(pd_entry_t pd, paddr_t phyDir, vaddr_t copyDir){
    mmu_map_page(   (uint32_t)pd, copyDir,
                    phyDir, MMU_U | MMU_W | MMU_P);
    copy_page(phyDir, (paddr_t)0xF151C000);
}
```

FUNCIONES AUXILIARES

Nos queda implementar las funciones auxiliares:

- `uint32_t task_selector_to_CR3(uint16_t selector);`
  - Nos permite encontrar el directorio de páginas de una tarea cualquiera en base a su task segment.

- `paddr_t virt_to_phy(uint32_t cr3, vaddr_t virt);`
  - Devuelve **la dirección física** asociada al cr3 y la dirección virtual pasadas por parámetro.
  - Esta funcion asume que existe un mapeo bajo la direccion **virt.**

FUNCIONES AUXILIARES

¿Cómo obtener el CR3?
Usamos el selector de segmento asociado a la tarea para indexar la GDT. Con el descriptor de la TSS podemos llegar a la estructura que necesitamos y de ella leer el valor del registro.

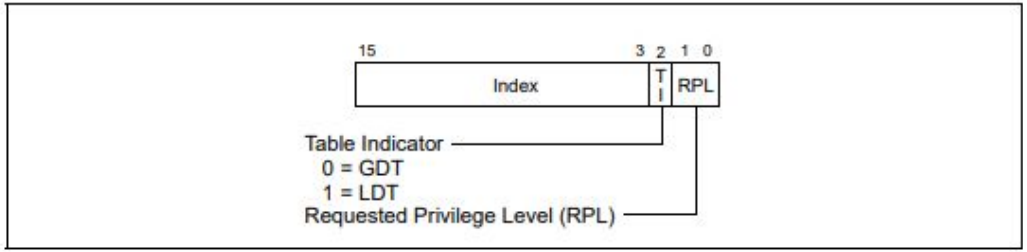


Figure 3-6. Segment Selector

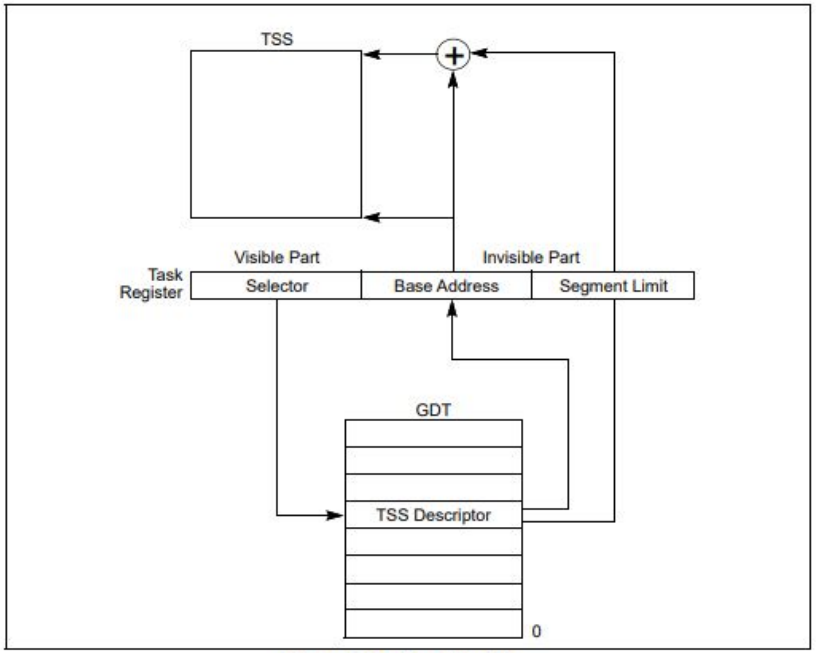


Figure 9-5. Task Register

FUNCIONES AUXILIARES

¿Cómo obtener el CR3?

Del descriptor obtenemos la base de la TSS juntando las partes baja, media y alta.

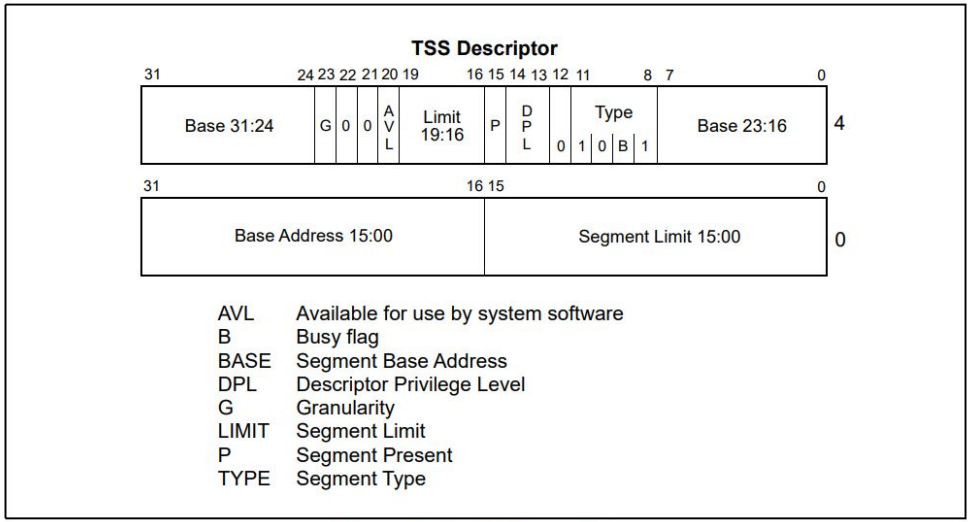


Figure 9-3. TSS Descriptor

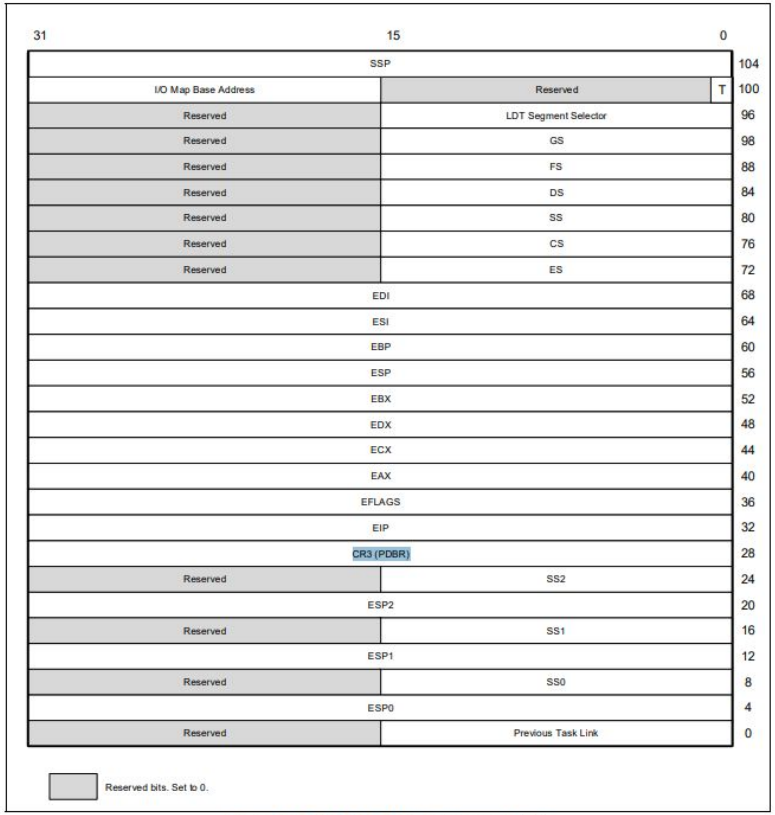


Figure 9-2. 32-Bit Task-State Segment (TSS)

FUNCIONES AUXILIARES

tasks.c

```
uint32_t task_selector_to_CR3(uint16_t selector) {
    uint16_t index = selector >> 3; // Sacamos los atributos
    gdt_entry_t* taskDescriptor = &gdt[index]; // Indexamos en la gdt
    tss_t* tss = (tss_t*)((taskDescriptor->base_15_0) |
                          (taskDescriptor->base_23_16 << 16) |
                          (taskDescriptor->base_31_24 << 24));

    return tss->cr3;
}
```

FUNCIONES AUXILIARES

¿Cómo traducir de virtual a física?

Necesitamos “seguir el camino” de la traducción usando la estructura de paginación:

`CR3 -> PD -> PD entry -> PT -> PT entry -> Pagina Física`

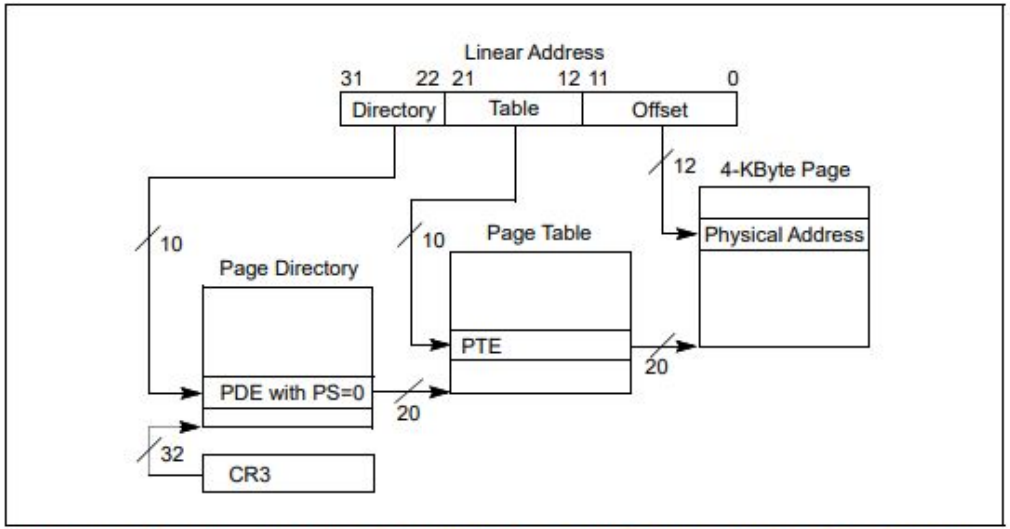


Figure 5-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

FUNCIONES AUXILIARES

tasks.c

```
paddr_t virt_to_phy(uint32_t cr3, vaddr_t virt) {

    uint32_t* cr3 = task_selector_to_cr3(task_id);

    uint32_t pd_index = VIRT_PAGE_DIR(virtual_address);
    uint32_t pt_index = VIRT_PAGE_TABLE(virtual_address);


    pd_entry_t* pd = (pd_entry_t*)CR3_TO_PAGE_DIR(cr3);


    pt_entry_t* pt = pd[pd_index].pt << 12;


    return (paddr_t) (pt[pt_index].page << 12);
}
```

EXTRA

EXTRA

Nuestro kernel "Orga Génesis" es funcional, pero ineficiente. Hemos detectado que algunas tareas de Nivel 3 solicitan acceso al dispositivo (opendevice) pero luego **nunca leen los datos**.

Esto es un desperdicio crítico de recursos. En particular:

- En **Modo Copia**, se gastan 4KB de RAM valiosa para una copia que no se usa.
- El deviceready (IRQ 40) gasta ciclos de CPU actualizando copias.

Para solucionar esto, implementaremos una nueva tarea de **nivel 0** llamada task_killer. El codigo de esta tarea reside en el area de kernel, se ejecutará como las demas tareas en round robin y debera deshabilitar cualquier tarea de nivel 3 que haya desperdiciado recursos por mucho tiempo.

Una tarea se considera *ociosa* y debe ser deshabilitada si cumple **todas** las siguientes condiciones:

1. Ha estado en estado TASK_RUNNABLE por más de 100 "ticks" de reloj.
2. Tiene un acceso *activo* al dispositivo (es decir, task[i].mode != NO_ACCESS y task[i].status != TASK_BLOCKED).
3. **No ha leído** la memoria del buffer (DMA o Copia) desde la última vez que el "killer" la inspeccionó.

EXTRA

3. Describa todos los pasos para crear e inicializar la nueva tarea `task_killer`:
Detalle los cambios necesarios en:

1. **GDT:** ¿Cómo se crea la nueva entrada de TSS (Task State Segment) en la GDT?
2. **TSS:** ¿Qué campos *críticos* de la `struct tss` (TSS) de esta nueva tarea debe inicializar? (Mencione al menos `EIP`, `CS`, `DS`, `ESP0`, `SS0` y `CR3`).
3. **Scheduler:** ¿Cómo se agrega esta nueva tarea Nivel 0 al array `sched_tasks`?

PASOS A REALIZAR

(Estructuras) Debemos:

- Agregar una entrada en la GDT con la TSS de la nueva tarea a ejecutar.
- Agregar una entrada de tarea en el scheduler.
- Agregar alguna estructura adicional para que se vayan contando los ticks que pasan.

(Funciones):

- Tenemos que implementar el código para la tarea, la cual ejecuta siempre el mismo código y permite “matar” a aquellas tareas que se aprovechen de la CPU.
- Tenemos que modificar la rutina del clock para que cuente la cantidad de ciclos que pasaron para cada tarea.

ESTRUCTURAS + FUNCIONES

Afortunadamente, tenemos unas funciones que nos permiten crear una tarea de nivel 3, inicializando todas las estructuras de una nueva tarea. Podemos aprovecharnos de este código y, con algunas modificaciones, crear una funcion que cree tareas de nivel 0.

tasks.c

```
static int8_t create_task(tipo_e tipo) {
  size_t gdt_id;

  for (gdt_id = GDT_TSS_START; gdt_id < GDT_COUNT;
gdt_id++)
    if (gdt[gdt_id].p == 0) break;

  kassert(gdt_id < GDT_COUNT,
      "No hay entradas disponibles en la GDT");

  int8_t task_id = sched_add_task(gdt_id << 3);
  tss_tasks[task_id] =
      tss_create_user_task(task_code_start[tipo]);

  gdt[gdt_id] =
      tss_gdt_entry_for_task(&tss_tasks[task_id]);
  return task_id;
}
```

tss.c

```
tss_t tss_create_user_task(
                    paddr_t code_start) {
  return (tss_t) {
    .cr3 = mmu_init_task_dir(code_start),
    .esp = TASK_STACK_BASE + PAGE_SIZE,
    .ebp = TASK_STACK_BASE + PAGE_SIZE,
    .eip = TASK_CODE_VIRTUAL,
    .cs = GDT_CODE_3_SEL,
    .ds = GDT_DATA_3_SEL,
    .es = GDT_DATA_3_SEL,
    .fs = GDT_DATA_3_SEL,
    .gs = GDT_DATA_3_SEL,
    .ss = GDT_DATA_3_SEL,
    .ss0 = GDT_DATA_0_SEL,
    .esp0 = mmu_next_free_kernel_page() +
PAGE_SIZE,
    .eflags = EFLAGS_IF,
  };
}
```

ESTRUCTURAS + FUNCIONES

Afortunadamente, tenemos unas funciones que nos permiten crear una tarea de nivel 3, inicializando todas las estructuras de una nueva tarea. Podemos aprovecharnos de este código y, con algunas modificaciones, crear una funcion que cree tareas de nivel 0.

tasks.c

```
static int8_t create_task_kill() {
  size_t gdt_id;
  for (gdt_id = GDT_TSS_START; gdt_id < GDT_COUNT;
gdt_id++) {
    if (gdt[gdt_id].p == 0) break;
  }

  kassert(gdt_id < GDT_COUNT,
      "No hay entradas disponibles en la GDT");

  int8_t task_id =
    sched_add_task(gdt_id << 3);

  tss_tasks[task_id] =
    tss_create_kernel_task(&killer_main_loop);

  gdt[gdt_id] =
    tss_gdt_entry_for_task(&tss_tasks[task_id]);
  return task_id;
}
```

tss.c

```
tss_t tss_create_kernel_task(paddr_t code_start)
  vaddr_t stack = mmu_next_free_kernel_page();
  return (tss_t) {
    .cr3 = create_cr3_for_kernel_task(),
    .esp = stack + PAGE_SIZE,
    .ebp = stack + PAGE_SIZE,
    .eip = (vaddr_t)code_start,
    .cs = GDT_CODE_0_SEL,
    .ds = GDT_DATA_0_SEL,
    .es = GDT_DATA_0_SEL,
    .fs = GDT_DATA_0_SEL,
    .gs = GDT_DATA_0_SEL,
    .ss = GDT_DATA_0_SEL,
    .ss0 = GDT_DATA_0_SEL,
    .esp0 = stack + PAGE_SIZE,
    .eflags = EFLAGS_IF,
  };
}
```

ESTRUCTURAS + FUNCIONES

El esquema de paginación para la tarea task_kill solo necesita tener un Identity Mapping del área de kernel. Esto es porque:

- El código de la tarea vive en el área de kernel, y por lo tanto es accesible desde el Identity mapping
- El stack de la tarea tambien vive en el area libre de kernel (Por ser una pagina sacada del area libre de kernel), y ademas mantiene los atributos de escritura (No van a surgir page faults por escribir en una pagina Read-Only).

mmu.c

```
paddr_t create_cr3_for_kernel_task() {
  // Inicializamos el directorio de paginas
  paddr_t task_page_dir = mmu_next_free_kernel_page();
  zero_page(task_page_dir);

  // Realizamos el identity mapping
  for (uint32_t i = 0; i < identity_mapping_end; i += PAGE_SIZE) {
    mmu_map_page(task_page_dir,i, i, MMU_W);
  }
  return task_page_dir;
}
```

RUTINA DEL CLOCK

isr.asm

```
extern add_tick_to_task
global _isr32
_isr32:
    pushad

    call pic_finish1
    call next_clock

    call add_tick_to_task

    call sched_next_task

    ; ...

    iret
```

sched.c

```
// Tarea actualmente en ejecución
int8_t current_task = 0;

static uint8_t contador_de_ticks[MAX_TASKS] =
{0};

void add_tick_to_task() {
     contador_de_ticks[current_task]++;
}
```

CODIGO - TASK_KILLER

tasks.c

```
void killer_main_loop(void) {
   while (true) {
    for (int i = 0; i < MAX_TASKS; i++) {
      if (i == current_task) continue;
      sched_entry_t* task = &sched_tasks[i];

      if (task->status == TASK_PAUSED) continue;
      if (task->mode != NO_ACCESS && task->status != TASK_BLOCKED) continue;
      if (task_ticks[i] <= 100) continue;

      vaddr_t vaddr = 0;

       if (task->mode == ACCESS_DMA) {
         vaddr_to_check = VADDR_DMA;
      } else {
         vaddr_to_check = task->copyDir;
      }

      pte_t* pte = mmu_get_pte_for_task(task->selector, vaddr);
      if (pte->accessed == 0) {
        task->status = TASK_KILLED;
      } else {
        tasks_ticks[i] = 0;
        pte->accessed = 0;
      }
    }
  }
}
```

CODIGO - TASK_KILLER

mmu.c

```
pt_entry_t* mmu_get_pte_for_task(uint16_t task_id, vaddr_t virtual_address) {

    uint32_t* cr3 = task_selector_to_cr3(task_id);

    uint32_t pd_index = VIRT_PAGE_DIR(virtual_address);
    uint32_t pt_index = VIRT_PAGE_TABLE(virtual_address);


    pd_entry_t* pd = (pd_entry_t*)CR3_TO_PAGE_DIR(cr3);


    pt_entry_t* pt = pd[pd_index].pt << 12;


    return (pt_entry_t*) &(pt[pt_index]);
}
```