AYOC 2025, 1er cuatrimestre, 2do parcial. System programming. - Aprobado sin comentarios.

Resolución

El enunciado nos pide definir en primera instancia una syscall `malloco`. Dado que es una syscall estamos hablando de una interrupción que puede ser ejecutada por la tarea. Para realizar el registro de la interrupción, cargamos la entrada en la IDT:

```
IDT_ENTRY3(90);
```

La definición de `IDT_ENTRY3` es la siguiente:

```
#define IDT_ENTRY3(numero)                                          \
  idt[numero] = (idt_entry_t) {                                     \
    .offset_31_16 = HIGH_16_BITS(&_isr##numero),                    \
    .segsel = GDT_CODE_0_SEL,                                       \
    .offset_15_0 = LOW_16_BITS(&_isr##numero),                      \
    .type = INTERRUPT_GATE_TYPE,                                    \
    .dpl = 0x3,                                                     \
    .present = 1                                                    \
  }
```

Observemos que los bits correspondientes al `dpl` tienen valor 0b11, por lo que permitiremos que la interrupción sea lanzada por la tarea (que asumimos tiene el mismo nivel de privilegio). A continuación mostramos un ejemplo de como esto sucede junto con el protocolo.

```
    ; tarea ejecutándose

    mov eax, <cantidad_de_bytes_a_allocar>
    int 90

    ; en este momento eax = dirección virtual donde la memoria está reservada.
```

Teniendo en cuenta las condiciones pedidas en el enunciado, definimos el handler para la interrupción:

```
    ; malloco
```

```
    global _isr90
    _isr90:
        pushad

            ; derivaremos la lógica principal a una función en c, malloco_c.
            ; entre otras cosas, la función ajustará la pila para que al terminar la interrupción
            ; la tarea encuentre en el registro eax el valor correspondiente al resultado de la syscall
            ; -> esto se debe a que al ejecutar popad, los registros de propósito general volverán a tener
            ; -> los valores originales al momento de interrumpir la tarea, en el caso del registro eax, el tamaño de bytes a
            ; -> reservar según el protocolo. queremos devolver en eax la dirección asignada

            push eax
            push esp
            call malloco_c
            add esp, 8

        popad
        iret
```

```
// 4mb / 4kb: vamos a asignar páginas de 4kb
#define MAX_PAGES_BY_TASK 1000

typedef struct reservation {
    uint8_t task_id;
    uint32_t size;
    vaddr_t assigned_virtual_addr;
    bool invalid;
} __attribute__((__packed__, aligned(8))) reservation_t;

static uint32_t reservations_size = 0;

// según el enunciado no nos preocuparemos por que el array tenga el espacio suficiente para registrar todas las reservas
// por esa razón MAX_RESERVATIONS_ALLOWED no está definido
static reservation_t reservations[MAX_RESERVATIONS_ALLOWED] = {0};

static uint32_t pages_by_task[MAX_TASKS] = {0};

// busca la reserva de memoria más actual en nuestra estructura para una tarea
reservation_t* get_last_reservation(uint8_t task_id) {
```

```
    reservation_t* result = NULL ;

    for(int i = reservations_size - 1; i >= 0; i--) {
        if(reservations[i].task_id = task_id && !reservations[i].invalid) {
            result = &reservations[i];
            break;
        }
    }

    return result;
}

void assign_reservation(uint8_t task_id, uint32_t size, vaddr_t virtual ) {
    reservations[reservations_size] = (reservations_t) { task_id, size, virtual, false }

    reservations_size += 1;
}

void malloco_c(uint32_t* esp, uint32_t size) {

    /**

    Estado actual del stack (más cosas se van a ir agregando a medida que se ejecute la función,
    pero no importa pues tenemos esp como parámetro que nos da un punto de referencia a partir de
    donde empezar a buscar valores)

    direcciones bajas ---------------

        esp ->  |       |   offset +0
                |DIR RET|   offset +4
                |ESP    |   offset +8
                |SIZE   |   offset +12
                |EDI    |   offset +16
                |ESI    |   offset +20
                |EBP    |   offset +24
                |ESP    |   offset +28
                |EBX    |   offset +32
                |EDX    |   offset +36
                |ECX    |   offset +40
                |EAX    |   offset +44

    direcciones altas ---------------
```

```
    orden pushad EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI

    **/

    bool another_page_allowed = pages_by_task[current_task] <= MAX_PAGES_BY_TASK;

    if(!another_page_allowed) {
        // acomodar eax -> null ; *(esp + 4 * 11) = NULL
        esp[11] = NULL;
        return;
    }

    // obtenemos la asignación de memoria más reciente (es la vigente)
    reservation_t* reservation = get_last_reservation(current_task); // current_task: sched.c

    if(
        reservation == NULL ||                  // no hay última asignación,
        reservation->size + size > PAGE_SIZE    // espacio insuficiente en esa asignación ; PAGE_SIZE: defines.h
    ) {
        // nueva virtual addr
        vaddr_t virtual = (reservation == NULL) ? ((vaddr_t)(0xA10C0000)) : (reservation->assigned_virtual_addr + 0x1000);

        // creamos una asignación nueva
        assign_reservation(current_task, size, virtual);
        esp[11] = virtual;
        pages_by_task[current_task] += 1;
        return;
    }

    // espacio suficiente en esa asignación
    reservation->size += size;
    esp[11] = reservation->assigned_virtual_addr;
}
```

Al haber ejecutado la syscall, la memoria se asigna en estructuras pero no queda mapeada a la tarea. Para terminar de implementar el comportamiento lazy pedido, asignaremos (mapearemos) la memoria cuando esta sea accedida. Cuando una tarea intenta acceder a memoria no mapeada (teniendo paginación activado), se produce una excepción de tipo Page Fault. La idea es verificar en el handler de esa excepción si la dirección que está siendo accedida está en rango de alguna reservada. Si ese fuera el caso, mapeariamos el espacio reservado acorde al enunciado. Si no fuera el caso desalojaremos la tarea.

Esta parte del enunciado es muy similar a un ejercicio del TP de System Programming. La idea es hacer algo similar.

La excepción de tipo Page Fault se corresponde con el índice 14 en el vector de interrupciones. Según el TP que estuvimos haciendo, las excepciones terminan llamando eventualmente a una función llamada kernel_exception desde la cual haremos la verificación pedida. Notemos que estoy asumiendo fuertemente la estructura del TP para esta parte del ejercicio.

```
bool kernel_exception(control_regs cregs,
                      segment_regs sregs, general_regs gregs,
                      kernel_error_frame frame) {
  //breakpoint();

  // esta parte es importante, acá solamente estoy copiando y pegando parte del trabajo práctico.
  if(frame.exception_number == 14) return page_fault_handler(cregs.cr2);

  print_exception_template();
  print(code2exception[frame.exception_number], 20, 2, 0x0A);

  sregs.cs = frame.cs;
  print_regs(&gregs, &sregs, &cregs, frame.eip, frame.eflags, frame.error_code);

  print_hex( *((uint32_t*)gregs.esp), 8, STACK_START_COL, 21, C_BG_BLACK | C_FG_LIGHT_GREEN);
  print_hex(*((uint32_t*)gregs.esp + 1), 8, STACK_START_COL, 23, C_BG_BLACK | C_FG_LIGHT_GREEN);
  print_hex(*((uint32_t*)gregs.esp + 2), 8, STACK_START_COL, 25, C_BG_BLACK | C_FG_LIGHT_GREEN);

  return false;
}


bool page_fault_handler(vaddr_t virt) {
  print("Atendiendo page fault...", 0, 0, C_FG_WHITE | C_BG_BLACK);
  // Chequeemos si el acceso fue para una dirección reservada
  // En caso de que si, mapear la pagina

  if(in_reserved_range(virt)) return false;

  uint32_t cr3 = rcr3();

  paddr_t free_user_page = mmu_next_free_user_page();

  mmu_map_page(cr3, virt, free_user_page, PTE_ATTR_PRESENT | PTE_ATTR_WRITE | PTE_ATTR_USER);
```

```
  return true;
}

bool in_reserved_range(vaddr_t virt) {
    // en esta función revisamos si hay memoria reservada para la tarea
    // notemos que en malloco_c devolvemos vaddr_t alineadas a 4k
    // sabemos que cada página que asignaremos tiene tamaño 4k
    // tenemos que revisar si el inicio de la página en donde se encuentra la dirección virtual virt está disponible
    // según la estructura que definimos para reservar memoria

    // una dirección virtual tiene la siguiente estrutura:
    // virt = 10 bits idx pd | 10 bits idx pt | 12 bits offset

    // los 20 bits más altos determinarán el inicio de la página (virtual), que es lo que tenemos que buscar
    // en nuestra estructura.

    vaddr_t candidate_page_address = virt & (0xFFFFF000);

    // ahora buscamos en la estructura si para la tarea actual existe alguna "reserva", si es así entonces estamos bien.

    reservation_t* result = NULL ;

    for(int i = reservations_size - 1; i >= 0; i--) {
        if(
          reservations[i].task_id = current_task &&
          rservations[i].assigned_virtual_addr == candidate_page_address &&
          !rservations[i].invalid
        ) {
            return true;
        }
    }

    return false;

    // observemos lo siguiente (solo un detalle):
    // qué pasa si la reserva que encontramos no tiene espacio suficiente para el dato que se quiere mirar?
    // si ese fuera el caso: previamente se pudo leer/escribir datos en la página, por lo que estaba mapeada, por lo que no
    // hubiera habido un page fault. no estariamos corriendo esta función.
}
```

Ahora: cómo desalojar a la tarea en caso de que haya querido acceder a una dirección de memoria inválida? Observemos que la función `kernel_exception` devuelve un booleano indicando si se pudo salvar o no la excepción. Esto es similar también a lo que se pedia para el TP de System Programming. Para mostrar como se desaloja la tarea vamos a ir un paso hacia atrás en la llamada a `kernel_exception` y a mostrar como es el comportamiento en la _isr14 que lo llama.

Esta es la definición de la macro que permite registrar excepciones en la IDT. Está todo copiado para dar contexto. La parte que nos interesa está casi al final.

```
; ISR that pushes an exception code.
%macro ISRE 1
global _isr%1

_isr%1:
  ISRc %1
  add esp, 4
  iret ; <- PARCIAL: Esto es imporante, fue movido respecto al proyecto base.
%endmacro


;; Definición de MACROS
;; ----------------------------------------------------------------------- ;;

%macro ISRc 1
    push DWORD %1
    ; Stack State:
    ; [ INTERRUPT #] esp
    ; [ ERROR CODE ] esp + 0x04
    ; [ EIP        ] esp + 0x08
    ; [ CS         ] esp + 0x0c
    ; [ EFLAGS     ] esp + 0x10
    ; [ ESP        ] esp + 0x14 (if DPL(cs) == 3)
    ; [ SS         ] esp + 0x18 (if DPL(cs) == 3)

    ; GREGS
    pushad
    ; Check for privilege change before anything else.
    mov edx, [esp + (8*4 + 3*4)]

    ; SREGS
    xor eax, eax
```

```
    mov ax, ss
    push eax
    mov ax, gs
    push eax
    mov ax, fs
    push eax
    mov ax, es
    push eax
    mov ax, ds
    push eax
    mov ax, cs
    push eax

    ; CREGS
    mov eax, cr4
    push eax
    mov eax, cr3
    push eax
    mov eax, cr2
    push eax
    mov eax, cr0
    push eax

    cmp edx, CS_RING_0_SEL
    je .ring0_exception

    ; COMPLETAR (opcional):
    ;   Si caemos acá es porque una tarea causó una excepción
    ;   En lugar de frenar el sistema podríamos matar la tarea (o reiniciarla)
    ;   ¿Cómo harían eso?
    call kernel_exception
    add esp, 10*4
    popad

    xchg bx, bx
    jmp $

; PARCIAL: ESTA ES LA PARTE QUE NOS INTERESA!!! --------------------------------
; No se si los offsets están bien. Hice la copia directamente del archivo del TP y modifiqué algunas cosas.
; Para ilustrar la solución a lo que pide el enunciado.
.ring0_exception:
    call kernel_exception
```

```
    cmp al, 1
    je .done_ok ; Si se pudo resolver la excepción, estamos bien y seguimos normalmente

    add esp, 10*4

    ; No se pudo resolver la excepción.
    ; Marcamos la tarea.

    call sched_pause_current_task
    call free_current_task

    ; Hacemos el salto a la siguiente tarea disponible.
    ; Realizamos el cambio de tareas en caso de ser necesario
    call sched_next_task
    str bx
    cmp ax, bx
    je .done_ok

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

.done_ok:
    add esp, 11*4
    popad

%endmacro
```

Como se puede ver, el salto es practicamente el mismo que hace el handler del clock. En este caso depende de la operación para salvar la excepción.

Definamos las funciones auxiliares:

```
void sched_pause_current_task() {
  sched_tasks[current_task].state = TASK_PAUSED;
}

// esta función tiene por objetivo invalidar o marcar las areas de memoria reservadas mediante malloco
void free_current_task() {
    for(int i = reservations_size - 1; i >= 0; i--) {
        if(reservations[i].task_id = current_task) {
```

```
            reservations[i].invalid = true;
        }
    }
}
```

Para la segunda syscall que pide implementar el enunciado, `chau` haremos las definiciones análogas a `malloco` para el sistema:

```
IDT_ENTRY3(91);
```

Con el siguiente protocolo o ejemplo de uso:

```
    ; tarea ejecutándose

    mov eax, <dirección virtual del bloque a liberar>
    int 91

    ; deberá retornar asincronamente. no se esperan valores especiales en los registros
```

Implementaremos ahora el handler para la idt entry de la siguiente manera:

```
    ; chau

    global _isr91
    _isr91:
        pushad

            ; marcaremos la memoria que no se utilizará más mediante una funcion llamada free_current_task_block

            push eax
            call free_current_task_block
            add esp, 4

        popad
        iret
```

```
void free_current_task_block(vaddr_t virtual) {
```

```
    // por enunciado asumimos que virtual es la dirección base del bloque asignado, por lo que no seria necesario
    // hacerle transformaciones.

    for(int i = reservations_size - 1; i >= 0; i--) {
        if(reservations[i].task_id = current_task && reservations[i].assigned_virtual_addr == virtual) {
            reservations[i].invalid = true;
            break;
        }
    }
}
```

Hasta acá en caso de que querramos liberar la memoria asignada mediante malloco para una tarea, tenemos un mecanismo (syscall) para marcar dicha memoria. Tenemos que definir ahora la tarea "garbage collector" que la liberará. Esta tarea estará definida como las otras tareas en el sistema, la diferencia es que tiene privilegios de kernel.

```
// archivo de la tarea que libera la memoria, garbage collector
void task(void) {
        while (true) {

        for(int i = reservations_size - 1; i >= 0; i--) {
            if(reservations[i].invalid) {
                // sobre cada area de memoria que encontramos reservada y que esta invalidad
                // querriamos desmapearla de su respectiva tarea.
                // las estructuras utilizadas a continuación están definidas en el TP de System Programming.

                tss_t* tss_target = &tss_tasks[reservations[i].task_id];

                uint32_t target_cr3 = tss_target->cr3;

                // esta función fue definida en el TP de System Programming.
                mmu_unmap(target_cr3, reservations[i].assigned_virtual_addr);
            }
        }

        }
}
```

Para crear la tarea "garbage collector" definiremos las siguientes funciones similares a las que están definidas en el TP de SP, se cambian algunos parámetros para crear tareas de nivel 0.

```c
typedef enum {
  ...
  TIPO_GARBAGE_COLLECTOR = 4,
} tipo_e;

/**
 * Array que nos permite mapear un tipo de tarea a la dirección física de su
 * código.
 */
static paddr_t task_code_start[..] = {
    [TIPO_GARBAGE_COLLECTOR] = (0x0001C000)
};

uint32_t mmu_init_system_task_dir(paddr_t phy_start) {
  paddr_t phy_task_pd = mmu_next_free_kernel_page();

  uint32_t cr3 = (phy_task_pd & 0xFFFFF000) | CR3_TASK_ATTR;

  // 4mb / 4kb = 2^22 / 2^12 = 2^10 = 1024

  // mapeo del espacio del kernel (identity mapping)
  for(uint32_t i = 0; i < 1024; i++) {
    mmu_map_page(cr3, i * 0x1000, i * 0x1000, PTE_ATTR_PRESENT | PTE_ATTR_WRITE);
  }

  // mapeo del código
  mmu_map_page(cr3, CODE_VIRTUAL_PAGE, phy_start, PTE_ATTR_PRESENT); // NOTAR QUE NO TIENE EL ATTR DE USER

  return cr3;
}

tss_t tss_create_system_task(paddr_t code_start) {
  // notemos que al ser una tarea de nivel 0, no estamo seteando la pila de nivel 3
  // a diferencia de lo que hacemos en el TP de SP.

  uint32_t cr3 = mmu_init_system_task_dir(code_start);

  vaddr_t code_virt = CODE_VIRTUAL_PAGE;
  vaddr_t stack0 = mmu_next_free_kernel_page(); // el stack de nivel 0 lo ponemos en el espacio del kernel
  vaddr_t esp0 = stack0 + (0x1000 - 4);
```

```
  return (tss_t) {
    .cr3 = cr3,
    .eip = code_virt,
    .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 = esp0,
    .eflags = EFLAGS_IF,
  };
}

static int8_t create_system_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_system_task(task_code_start[tipo]);
  gdt[gdt_id] = tss_gdt_entry_for_task(&tss_tasks[task_id]);
  return task_id;
}

void tasks_init(void) {
  int8_t task_id;

  task_id = create_system_task(TIPO_GARBAGE_COLLECTOR);
  sched_enable_task(task_id);
}
```

Desde el kernel deberia llamarse a tasks_init y asumimos que el código de la tarea "garbage collector" estará en la dirección física 0x0001C000 (por configuración en el Makefile).

```
    call tasks_init
```

Para saltar a la tarea especial cada 100 ticks de reloj, mantendremos un contador en el scheduler, alterando nuestro algoritmo de round robin: si los 100 ticks son alcanzados, se devolverá la tarea "garbage collector" a ejecutar. No son necesarios cambios en el handler del clock: se mantiene la forma en la que se salta.

```
// por simplicidad, voy a asumir que el índice en la gdt de la tarea "garbage collector" es 20.

static uint8_t ticks = 0;

uint16_t sched_next_task(void) {

  ticks++;

  if(ticks-1 == 100) {
    ticks = 0;
    return 20 << 3; // será el registro TR de la tarea "garbage collector"
  }

  // DE ACA para abajo es la misma lógica que en el TP de SP

  // Buscamos la próxima tarea viva (comenzando en la actual)
  int8_t i;
  for (i = (current_task + 1); (i % MAX_TASKS) != current_task; i++) {
    // Si esta tarea está disponible la ejecutamos
    if (sched_tasks[i % MAX_TASKS].state == TASK_RUNNABLE) {
      break;
    }
  }

  // Ajustamos i para que esté entre 0 y MAX_TASKS-1
  i = i % MAX_TASKS;

  // Si la tarea que encontramos es ejecutable entonces vamos a correrla.
  if (sched_tasks[i].state == TASK_RUNNABLE) {
    current_task = i;
    return sched_tasks[i].selector;
  }
```

```
  // En el peor de los casos no hay ninguna tarea viva. Usemos la idle como
  // selector.
  return GDT_IDX_TASK_IDLE << 3;
}
```

Notas (agregadas al final después de repasar el enunciado):

- En la asignación de páginas fisicas, se asume que las mismas estarán inicializadas en 0.