(68) Nota: 100 (A)

Organización del Computador II - Departamento de Computación - UBA

Segundo Parcial

Corrigió: Ignacio

Primer Cuatrimestre 2023

Normas generales

- El parcial es INDIVIDUAL
- Puede disponer de la bibliografía de la materia y acceder al repositorio de código del taller de system programming, desarrollado durante la cursada
- Las resoluciones que incluyan código, pueden usar assembly o C. No es necesario que el código compile correctamente, pero debe tener un nivel de detalle adecuado para lo pedido por el ejercicio.
- Numere las hojas entregadas. Complete en la primera hoja la cantidad de hojas entregadas
- Entregue esta hoja junto al examen. La misma no se incluye en el total de hojas entregadas.
- Luego de la entrega habrá una instancia coloquial de defensa del examen

Régimen de Aprobación

- Para aprobar el examen es necesario obtener como mínimo **60 puntos**.
- Para promocionar es condición suficiente y necesaria obtener como mínimo **80 puntos** tanto en este examen como en el primer parcial

NOTA: Lea el enunciado del parcial hasta el final, antes de comenzar a resolverlo.

Enunciado

Ejercicio 1 - (70 puntos)

En un sistema similar al que implementamos en los talleres del curso (modo protegido con paginación activada), se tienen 5 tareas en ejecución y una sexta que procesa los resultados enviados por las otras. Cualquiera de estas 5 tareas puede en algún momento realizar una cuenta y enviar el resultado de la misma a la sexta tarea para que lo utilice de manera inmediata. Para ello la tarea que realizó la cuenta guardará el resultado de la misma en EAX. A continuación, la tarea que hizo la cuenta le cederá el tiempo de ejecución que le queda a la tarea que va procesar el resultado (lo recibirá en EAX). Tener en cuenta que la tarea que hizo la cuenta no volverá a ser ejecutada hasta que la otra tarea no haya terminado de utilizar el resultado de la operación realizada.

Se desea agregar al sistema una syscall para que la tareas después de realizar la cuenta en cuestión puedan cederle el tiempo de ejecución a la tarea que procesará el resultado.

Se pide:

a. Definir o modificar las estructuras de sistema necesarias para que dicho servicio pueda ser invocado.

b. Implementar la syscall que llamarán las tareas.

c. Dar el pseudo-código de la tarea que procesa resultados (no importa como lo procese).

d. Mostrar un pseudo-código de la función `sched_next_task` para que funcione de acuerdo a las necesidades de este sistema. Responder: ¿Qué problemas podrían surgir dadas las modificaciones al sistema? ¿Cómo lo solucionarías?

Se recomienda organizar la resolución del ejercicio realizando paso a paso los items mencionados anteriormente y explicar las decisiones que toman.

Detalles de implementación:

- Las 5 tareas originales corren en nivel 3.
- La sexta tarea tendrá nivel de privilegio 0.

Ejercicio 2 - (30 puntos)

Se desea implementar una modificación sobre un kernel como el de los talleres: en el momento de desalojar una página de memoria que fue modificada esta se suele escribir a disco, sin embargo se desea modificar el sistema para que no sea escrita a disco si la pagina fue modificada por una tarea específica.

Se les pide que hagan una función que, dado el CR3 de la tarea mencionada y la dirección física de la página a desalojar, diga si dicha pagina debe ser escrita a disco o no.

La función a implementar es:

```
uint8_t Escribir_a_Disco(int32_t cr3, paddr_t phy);
```

Detalles de implementación:

- Si necesitan, pueden asumir que el sistema tiene segmentación *flat*.
- NO DEBEN modificar las estructuras del kernel para llamar a la función que están creando. Solamente deben programar la función que se pide.

A tener en cuenta para la entrega (para todos los ejercicios):

- Está permitido utilizar las funciones desarrolladas en los talleres.
- Es necesario que se incluya una explicación con sus palabras de la idea general de las soluciones.
- Es necesario escribir todas las asunciones que haga sobre el sistema.
- Es necesaria la entrega de código o pseudocódigo que implemente las soluciones.

Entrego 5 hojas

1

2do parcial

1) a) Asumo que las 6 tareas fueron inicializadas en el scheduler (y que hicieron terminar)
O sea, cada una de ellas tiene una TSS y un descriptor correspondiente está en la GDT. Los descriptores de las 6 tareas de nivel 3 como las de nivel 0 tendrán los mismos atributos en el descriptor (DPL=0, S=0)

- Con esto, la TSS de la tarea de nivel 0 deberá tener en los selectores de segmento de código y datos de nivel 0 ~~[illegible]~~, (GDT_CODE_0_SEL y GDT_DATA_CODE_0_SEL). En nuestro sistema del taller no tenemos una función que inicialice tareas para que ejecuten a nivel 0, por lo que deberíamos crear esa función (hay que tener en cuenta que se debe pedir una página del kernel para el stack de la tarea)
- No hace falta ~~[illegible]~~ pedir una página del kernel para el stack de nivel 0 (pues ya una propia stack de tarea es de nivel 0) por lo que no haría un cambio de privilegio al invocar la interrupción, pues de que el campo esp0 de la su TSS puede tener cualquier valor.
- También cabe hacer que no haya una función que inicialice una estructura de paginación con páginas de código y datos de nivel 0 (supongo para mmu_init_task_dir ~~[illegible]~~ tareas crea estructuras de paginación ~~[illegible]~~ con páginas de código y datos de nivel 3
- ~~[illegible]~~ asumo que todas las tareas pueden ser inicializadas correctamente.
~~[illegible]~~ Creo una entrada en la IDT para la SYSCALL definiendo que su id es 47 (no se solapa con ninguna código de excepción ni ninguna otra interrupción en nuestro sistema)

Debo definir una entrada en la IDT con la macro:
IDT_ENTRY 3(47); //
En la inicialización de la IDT. Habría que debe ser una interrupción con DPL=3 para que pueda ser invocada por cualquier tarea a nivel de usuario.

b) La idea de la ISR es la siguiente:
Si se está ejecutando la tarea A, y el scheduler dijera que la siguiente tarea a ejecutar es la B

2

La tarea A hace la syscall, por lo que se debe deshabilitar la ejecución de la tarea A en el scheduler y se debe habilitar la ejecución de la tarea B. Además se guarda en la memoria el id de la tarea A, para que luego pueda volver a ser habilitada.
Una vez hecho eso se hace un jmp far al TR de la tarea B, para así lo que cause un cambio de contexto, guardando en el state de la tarea A en su TSS y restableciendo el contexto de la tarea B existente en su TSS, por lo que la ejecución continuará en el código de la tarea B.
Notar que:

- Cuando se cambia a la tarea B, para el scheduler se sigue ejecutando la tarea A dentro de su ciclo. Esto se hizo así para que cuando ocurra la interrupción de reloj durante la ejecución de la tarea B se cambie a la tarea B (que es la que le sigue a A), alterando el orden de ejecución de tareas.

- Cuando se vuelva a habilitar la tarea A su ejecución continuará en el popad de la isr47, luego se restauran los valores de sus registros y se vuelve a la ejecución de su código.

- Cuando se produzca la interrupción de reloj se guarda el contexto de la tarea B, por lo que esa ejecución continuará en ese punto. El estado de la tarea A no cambia.

- Al deshabilitar la tarea A, el scheduler la saltará al buscar la tarea que sigue al hacer sched_next_task, por lo que se saltará a su tarea siguiente.

- Ocurrirá lo mismo con la tarea B, pues el estar habilitada será ejecutada en si es la tarea siguiente en sched_next_task

```
global _isr47
_isr47:
    pushad ; Guarda los registros de propósito general de la tarea.
    push eax ; Pasa el parámetro para la función habilitar_tarea
    call habilitar_tarea_6
    pop eax
    jmp far [task_6_offset] ; // Salta al Call a la tarea 6
    popad                   ¿Cuál es el valor del selector?
    iret
```

Es de 32 bits por eso eax

```
uint8_t habilitar_tarea_6 (uint32_t resultado)
    // Uso current_task como variable global y task_6_id
    sched_disable_task(current_task);
    sched_enable_task(task_6_id);
    // Ahora accedo a la TSS de la tarea 6 para que en eax resida
    // el resultado de la tarea A
    // Busco el selector de la TSS de la tarea 6 en sched_entry_t, que tiene los descriptores de
    // todas las tareas
    uint16_t index_t6_tss = sched_entry_t[task_6_id] >> 3; // Ignoro los bits RPL y TI
    vaddr_t tss_t6_addr = (gdt[task_t6_id].base_31_24 <<24)
                        | (gdt[task_t6_id].base_23_16 <<16)
                        | (gdt[task_t6_id].base_15_0);
    tss_t* tss_t6 = (tss_t*) tss_t6_addr;
```

3

tss_t6->eax = resultado;

/* En todo el proceso anterior se accede al descriptor de TSS (que está en la GDT) para luego acceder a la TSS de la tarea 6 y cambiar el valor de su registro eax. */

tarea_desalojada = current_task // Es una variable global del scheduler

return current_task;

}

task_6_offset dd 0
task_6_selector dw 0

Esta sería la inicialización que señala la correción de la página anterior.
Me faltó detallar que task_6_selector es la parte de la memoria que guarda el TR de la tarea 6, para así poder hacer el cambio de tareas.
No es necesario darle algún valor a task_6_offset, pues será ignorado al hacer el jmp far.

Esta parte de la memoria será inicializada al iniciar la tarea 6, dentro de la memoria del kernel para que otra tarea no pueda manipularla

d) El kernel le da el control a la tarea 6, para ello el scheduler realiza un cambio de contexto. Al retornar su TSS, la tarea 6 tendrá en eax el resultado de la tarea A que la invocó. Esta tarea será inicializada en el estado de PAUSE por el scheduler para que no se ejecute hasta que sea requerida

TAREA 6.

while(True) {

Inicializa sus variables

Procesa el dato

// Una vez procesado el dato existe su ejecución
// y sigue con la siguiente tarea

Continúa en la otra hoja
Continúa ejecución

```
    //Habilito la tarea A
    sched_enable_task(tarea_desalojado);
    sched_disable_task(task_6_id);
    Cambiar_tarea();
  } //Cierro el ciclo while
}
```

En global Cambiar_tarea

```
Cambiar_tarea:
    pushad
    Call sched_next_task
    [illegible]
    [illegible]

    [illegible]
    jmp 0
    mov WORD [sched_task_selector], ax
    jmp far [sched_task_offset]
    popad
    ret
```

[illegible]

Todo el procedimiento de TAREA 6 se implica en un ciclo que no termina. La idea de esto es que, una vez la TAREA 6 terminó su ejecución, se quede en estado deshabilitada a sí misma (pues al haber terminado no puede seguir ejecutándose) y habilite la ejecución de la tarea A (que fue la que llamó la syscall) para luego hacer el cambio de contexto a la tarea

que le sigue en el scheduler.
Una vez hecho eso, si otra tarea B ejecuta la syscall leer_teclado y el estado de la tarea 6 en el popad de la función Cambiar_tarea, hace el ret y así volvió al principio del código que le correspondía a la tarea para que la tarea 6 pueda hacer lo que tenga que hacer con el resultado de la tarea B

Notas que:

- De esta manera la tarea 6 nunca termina, pero que al finalizar su procesamiento, al ver nuevamente de nuevo se vuelve al comienzo de la misma
- La tarea puede hacer uso del scheduler por ser de nivel 0.

En Cambiar_tarea se hace el salto muy similar a como lo hace la interrupción de reloj, se utilizan las variables de memoria de sched_task_selector y offset porque se ofrece su modificación en la interrupción de reloj, pero dicen podrían haberse utilizado diferentes variables de memoria

d) Como no fue necesario implementar la finalización de tareas, no hay hasta que Carlos en el sched_next_task.
Ahí es, se hace uso de la función sched_disable_task que ha se utiliza en los teclados y también se hace uso de sched_enable_task por fuera de la inicialización de las tareas

Se agregan las variables globales:

- task_6_id cuyo valor será asignado al tipo el id de la tarea 6
- tarea_desalojada que será cambiando su valor cuando una tarea cargue una syscall. Notar que se guarda en su id.

Los problemas que pueden surgir son:

- Que una tarea A llame a la syscall y la ejecución de la tarea 6 ha finalizado y otra tarea B llame a la syscall. Si esto pasa no hay manera de restablecer la ejecución de la tarea A porque se perdió el valor de tarea_desalojada. (para solucionar esto se podría implementar una cola de tareas, en la que se guarde en ella el id de la tarea y su resultado (para varias).
De esta manera, la tarea 6 hace uso de la cola utilizando tanto el id como el resultado del tope de la cola. al final del ciclo se desencola y solo termina su ejecución cuando la cola está vacía.

- Que la propia tarea 6 llame a la SYSCALL (ha heredado)

* Como nota, no hay cabio de privilegio cuando la tarea 6 es interrumpida por el reloj, pero no afecta en nada porque el iret sabe detectar el privilegio de la tarea interrumpida

(A) Con esto no se puede mantener el resultado dentro del TSS de la tarea 6, pero que al inicio del ciclo cargamos en eax el valor resultado de la cola.

Hice el ejercicio pensando que la página tenía que ser accedida, no modificada, así que cada vez que dije accedida tendría que haber escrito modificada.
La resolución del ejercicio no cambia mucho por eso.

5

2)

Para hacer esto tengo que ir recorriendo todo el directorio de página de la tarea, y de aquellas PT que sean válidas recorrer todas sus entradas de página para buscar si alguna de ellas tiene como dirección física la phy. ~~Para que cuando encuentre una página que tiene la dirección buscada~~

para luego chequear en el descriptor si fue accedido.
La función devolverá 1 si esa dirección física fue accedida y 0 sino

```
uint8_t Escribir_a_Disco(uint32_t cr3, paddr_t phy) {
    puint32_t pdAddr = (cr3 & 0xFFFFF000); //Ignoro los 12 bits bajos siguientes
    pd_entry_t* pd = (pd_entry_t*) pdAddr;
    uint8_t res = 0;
    for(int i=0; i<1024; i++) {
        pt_entry_t* pt = (pt_entry_t*) pd[i];
        // Me fijo si es una entrada valida (lo mismo si esta presente está en 1)
        if((pt->attr) & MMU_P) == 1) {
            res |= chequeo_pt(pt, phy);
        }
    }
    return res
}
```

Chequeo_pt hace cada una de las entradas de página, si una entrada es valida, se fija su dirección física, si coincide con phy, se fija si fue accedida

```
uint8_t Chequeo_pt(pt_addr_t* pt, paddr_t phy
    uint8_t res = 0
    for(int j=0; j<1024; j++) {
        if((pt[j]->attr & MMU_P) == 1) { // Chequeo validez
            if((pt[j]->page) << 12) == (phy & 0xFFFFF000)) { // Chequeo que sea la dirección física
                if((pt[j]->attr & MMU_A) == 1) { // Chequeo que haya sido accedida
                    res = 1;                    ¿Es ese el bit?
                    [illegible];
                }
            }
        }
    }
    return res;
}
```

Acá tendría que haber puesto MMU_D (o sea, chequear el bit dirty de la página)

Defino MMU_A = 1 << 5;

[illegible]

Notar que tanto la función principal podrían terminar su ejecución al encontrar que la página fue accedida, pero yo decidí elegir que terminara su ejecución notar también que podría haber tomado la dirección física que fue accedida, pues podría haber accedido desde otra dirección virtual? por eso no puedo asegurar que no fue accedida al recorrer toda la entrada.