PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2333 — Sistemas Operativos y Redes — 2/2024

Tarea 2

Lunes 30 de Septiembre

Fecha de Entrega: Miércoles 16 de Octubre a las 21:00
Composición: grupos de 2 personas

Objetivo

- Implementar una API para manejar el contenido de una memoria principal.

Contextualización

La tarea consiste en crear una API que maneja archivos en una memoria principal que está dividida en *frames*. La API deberá manejar la memoria de procesos simulados, lo que significa que estos procesos no se encuentran realmente en ejecución, pero si se les deberá asignar y liberar memoria como si lo estuvieran. Además, cada uno de estos procesos tendrá su propia memoria virtual paginada en la que tendrán archivos vinculados que representarán sus segmentos. Estos archivos presentarán una dirección virtual dentro de la memoria virtual del proceso, y deberá transformarlos a direcciones físicas, con ayuda de tablas de páginas.

Introducción

La paginación es un mecanismo que nos permite eliminar la fragmentación externa de la memoria y consiste en dividir la memoria virtual de un proceso en porciones de igual tamaño llamadas páginas. Mientras que, la memoria física es separada en porciones de igual tamaño llamados *frames*.

En este proyecto tendrán la posibilidad de experimentar con una implementación de un mecanismo de paginación simplificada sobre una memoria principal la cual será simulada por un archivo real. Deberán leer y modificar el contenido de esta memoria mediante una API desarrollada por ustedes. Se recomienda para este proyecto que vean los videos de **segmentación**, **paginación** y **paginación multinivel**.

Estructura de memoria `osrms`

El mecanismo de paginación a implementar será denominado `osrms`. La memoria física está representada por un archivo real del sistema que está organizada de la siguiente manera:

- Se divide en segmentos de tamaños fijos en el siguiente orden 8 KB, 128 B, 128 KB, 8 KB, 2 GB. El tamaño total de la memoria es la suma de los tamaños de estos segmentos.
- Los primeros 8 KB de la memoria corresponde a espacio reservado exclusivamente para la **Tabla de PCBs**.
- Le siguen 128 B destinados al **Bitmap Tablas de Páginas**.
- Los siguientes 128 KB de la memoria corresponde a espacio reservado exclusivamente para el **Espacio de Tabla de Páginas de Segundo Orden**.
- Luego, existen 8 KB destinados al **Frame Bitmap**.

- Los últimos 2 GB de memoria está dividido en conjuntos de Bytes denominados *frames*:
  - Tamaño de *frame*: 32 KB. La memoria contiene un total de 2^{16} *frames*.
  - Cada *frame* posee un numero secuencial que se puede almacenar en 16 bits, por lo cual se puede guardar en un `unsigned int`. **Este valor corresponde a su PFN**[1].

Además, cada proceso cuenta con un espacio virtual de memoria igual a 128 MB. Esta memoria virtual está dividida en páginas de 32 KB.

Tabla de PCBs. Se encuentra al inicio de la memoria y contiene información sobre los procesos. Está separada en 32 entradas, las cuales siguen la siguiente estructura:

- Tamaño de entrada: 256 Bytes.
- 1 Byte de estado. `0x01` si el proceso está en ejecución, o `0x00` en caso contrario.
- 1 Byte para indicar el `id` del proceso.
- 11 Bytes para indicar el nombre del proceso.
- 115 Bytes para una **Tabla de Archivos**.
- 128 Bytes para una **Tabla de Páginas de Primer Orden** del proceso.

Tabla de Archivos. Contiene la información de los archivos presentes en la memoria virtual de un proceso. Está separada por 5 entradas, donde cada una entrada cumple con el siguiente formato:

- Tamaño de entrada: 23 Bytes.
- 1 Byte de validez. `0x01` si la entrada es válida, o `0x00` en caso contrario.
- 14 Bytes para nombre del archivo.
- 4 Bytes para tamaño del archivo. El tamaño máximo de un archivo es de 64 MB.
- 4 Bytes para la dirección virtual. 5 bits no significativos `(0b00000)` + 12 bits VPN + 15 bits offset.

Tabla de Páginas de Primer Orden. Contiene información necesaria para acceder a las tabla de páginas de segundo orden de un proceso a partir de los primeros 6 bits del `VPN` de una dirección virtual. Está separada en 64 entradas, donde cada entrada sigue el siguiente formato:

- Tamaño de entrada: 2 Bytes.
- Cada entrada corresponde a un número entre 0 y 1023, e indica el número de la Tabla de Páginas de Segundo Orden.

Espacio de Tabla de Páginas de Segundo Orden. Tiene un tamaño de 128 KB y en esta se encuentran todas las Tabla de Páginas de Segundo Orden. Cada una de estas tablas tiene un tamaño de 128 Bytes, y pueden ser enumeradas del 0 al 1023.

Bitmap de Tablas de Páginas. Cuenta con un tamaño de 128 B (1024 bits). Cada bit indica si una tabla de páginas de segundo orden esta libre (0) o no (1). Por ejemplo, si el primer bit de Bitmap de Tablas de Páginas es 1, quiere decir que la primera tabla del Espacio de Tabla de Páginas de Segundo Orden esta siendo utilizada por algún proceso.

Tabla de Páginas de Segundo Orden. Contiene la información para traducir direcciones virtuales a físicas dentro de la memoria. Las direcciones físicas que se obtengan serán relativas a los últimos 2 GB de la memoria, es decir, si la dirección física relativa es dir entonces la absoluta será $8 \cdot 2^{10} + 128 + 128 \cdot 2^{10} + 8 \cdot 2^{10} + dir$ [2]. Una Tabla de Páginas de Segundo Orden está separada en 64 entradas, las cuales tienen la siguiente estructura:

---

1 Ese número no se encuentra guardado en la memoria física

2 En otras palabras, $|$tabla de PCBs$|+|$Bitmap de Tablas de Páginas$|+|$Espacio de Tabla de Páginas de Segundo Orden$|+|$Frame bitmap$|+dir$

- Tamaño de entrada: 2 Bytes.
- Cada entrada corresponde a un número entre 0 y 65535, correspondiente a un `PFN`.

Para calcular una dirección física a partir de una dirección virtual se deben seguir los siguientes pasos:

1. Obtener de la dirección virtual los 12 bits del `VPN` y 15 bits del `offset`.
2. Dirigirse a la entrada de la **Tabla de Páginas de Primer Orden** indicado por lo primeros 6 bits del `VPN` y obtener el número de la **Tabla de Páginas de Segundo Orden** indicado por la entrada.
3. Dirigirse a la entrada de la **Tabla de Páginas de Segundo Orden**(`SPTN`) indicada por los últimos 6 bits del `VPN` y obtenemos el `PFN`. Notar que la dirección absoluta de esta entrada estará dada por $8 \cdot 2^{10} + 128 + 128 \cdot$ `SPTN` + 2 $\cdot$`VPN`$[6:]$[3].
4. Luego, la dirección física será igual a `PFN` seguido del `offset`.

Frame bitmap. Cuenta con un tamaño de 8 KB (65536 bits). Cada bit del *Frame bitmap* indica si un *frame* está libre (0) o no (1). Por ejemplo, si el primer bit del *frame bitmap* tiene valor 1, quiere decir que el primer *frame* de la memoria esta siendo utilizado. En resumen:

- El *Frame bitmap* contiene un bit por cada *Frame* de la memoria principal.
- El *Frame bitmap* debe reflejar el estado actual de la memoria principal.

Frames. Aquí se almacenan los datos de los archivos. El tamaño de cada *frame* es de 32 KB. Se encuentran en los últimos 2 GB y en total hay 2^{16} *frames*.

Es importante destacar que la lectura y escritura de datos **deben** ser realizadas en orden ***little endian***. Consideren que el *endianess* solo afecta a los tipos de datos que tengan un tamaño mayor a un Byte, por lo que para este proyecto deben tenerlo en cuenta para los valores numéricos (direcciones, tamaños, etc). Además, si utilizan los tipos de datos correctos no deberían *swapear* los Bytes, ya que en general los computadores trabajan con ***little endian***.

API de `osrms`

Para poder manipular los archivos de los procesos, tanto para escritura como para lectura, deberá implementar una biblioteca que contenga las funciones necesarias para operar sobre la memoria principal. La implementación de la biblioteca debe escribirse en un archivo de nombre `osrms_API.c` y su interfaz (declaración de prototipos) debe encontrarse en un archivo de nombre `osrms_API.h`. Además, dentro de `osrms_API.c` se debe definir un `struct` llamado `osrmsFile`[4], la cual representará un *archivo abierto* y será utilizada para manejar los archivos pertenecientes a cada proceso.
Para probar su implementación debe escribir un archivo (por ejemplo, `main.c`) con una función `main` que incluya el *header* `osrms_API.h` y que utilice las funciones de la biblioteca para operar sobre un archivo que represente una memoria principal que debe ser recibido por la línea de comandos.

La biblioteca debe implementar las siguientes funciones:

Funciones generales

- `void os_mount(char* memory_path)`. Función para montar la memoria. Establece como variable global la ruta local donde se encuentra el archivo `.bin` correspondiente a la memoria.

[3] Esto se refiere a los últimos 6 bits del VPN.
[4] Utilice `typedef` para renombrar la estructura

- `void os_ls_processes()`. Función que muestra en pantalla los procesos en ejecución en la memoria.
- `int os_exists(int process_id, char* file_name)`. Función que verifica si un archivo con nombre `file_name` existe en la memoria del proceso con id `process_id`. Retorna `1` si existe y `0` en caso contrario.
- `void os_ls_files(int process_id)`. Función para listar los archivos dentro de la memoria del proceso. Imprime en pantalla los nombres y tamaños de todos los archivos presentes en la memoria del proceso con id `process_id`.
- `void os_frame_bitmap()`. Imprime el estado actual del *Frame Bitmap*. También debe imprimir el conteo de frames ocupados y libres.
- `void os_tp_bitmap()`. Imprime el estado actual del *Bitmap de Tablas de Páginas*. También debe imprimir el conteo de tablas de páginas de segundo orden ocupadas y libres.

Funciones para procesos

- `void os_start_process(int process_id, char* process_name)`. Función que inicia un proceso con id `process_id` y nombre `process_name`. Guarda toda la información correspondiente en una entrada en la **tabla de PCBs**. Cualquier cambio producido debe verse reflejado en memoria montada.
- `void os_finish_process(int process_id)`. Función para terminar un proceso con id `process_id`. Es importante que antes de que el proceso termine se debe liberar toda la memoria asignada a éste y no debe tener entrada válida en la **tabla de PCBs**. Cualquier cambio producido debe verse reflejado en memoria montada.

Funciones para archivos

- `osrmsFile* os_open(int process_id, char* file_name, char mode)`. Función para abrir un archivo perteneciente a `process_id`. Si `mode` es `'r'`, busca el archivo con nombre `file_name` y retorna un `osrmsFile*` que lo representa. Si `mode` es `'w'`, se verifica que el archivo no exista y se retorna un nuevo `osrmsFile*` que lo representa. En cualquier otro caso, la función debe retornar un puntero `NULL`.
- `int os_read_file(osrmsFile* file_desc, char* dest)`. Función para leer archivos. Lee un archivo desde descrito por `file_desc` y se crea una copia del archivo en la ruta indicada por `dest` dentro de su computador. Retorna la cantidad de Bytes leídos.

  El contenido de un archivo puede estar escrito en más de un *frame*, por lo que cuando se ha leído todo el contendido de un archivo de dicho *frame*, se debe continuar la lectura desde el principio del *frame* asociado a la siguiente página, y continuar repitiendo esto hasta leer completamente el archivo.
- `int os_write_file(osrmsFile* file_desc, char* src)`. Función para escribir archivos. Toma el archivo ubicado en la ruta `src` de su computador, y lo escribe dentro de la memoria montada. Es importante que cualquier cambio producido debe verse reflejado tanto en la memoria montada, como en `file_desc`. Finalmente, esta función retorna la cantidad de Bytes escritos.

  La escritura comienza desde el primer espacio libre dentro de la memoria virtual, por lo tanto, no necesariamente comenzarán a escribirse desde el inicio de una página. Esto significa que los archivos pueden compartir el mismo **frame** y **página**.

  Finalmente, la escritura se debe detener cuando:

  - No quedan *frames* disponibles para continuar, ó
  - Se termina el espacio disponible de la memoria virtual.
- `void os_close(osrmsFile* file_desc)`. Función para cerrar archivo. Cierra el archivo indicado por `file_desc`.

Ejecución

Para probar su biblioteca, debe usar un programa `main.c` que reciba un archivo que represente la memoria física (ej: `mem.bin`). El programa `main.c` deberá usar las funciones de la biblioteca `osrms_API.c` para ejecutar algunas instrucciones que demuestren el correcto funcionamiento de éstas. Una vez que el programa termine, todos los cambios efectuados sobre la memoria virtual deben verse reflejados en el archivo recibido.
La ejecución del programa principal debe ser:

```
valgrind ./osrms mem.bin
```

Donde `mem.bin` es el nombre del archivo que representa la memoria.
Por otra parte, un ejemplo de una secuencia de instrucciones que puede encontrarse en `main.c` es el siguiente:

```
os_mount(argv[1]); // Se monta la memoria.
```

Al terminar de ejecutar todas las instrucciones, el archivo `mem.bin` debe reflejar todos los cambios aplicados. Para su implementación, puede ejecutar todas las instrucciones dentro de las estructuras definidas en su programa y luego escribir el resultado final en la memoria o bien aplicar cada comando de forma directa en `mem.bin` de forma inmediata. Lo importante es que el estado final de la memoria virtual sea consistente con la secuencia de instrucciones ejecutada.

Para probar las funciones de su API, se hará entrega de dos memorias:

- `memformat.bin`: Memoria formateada. No posee procesos en "ejecución" ni archivos, y todo su bitmap está desocupado.
- `memfilled.bin`: Memoria con procesos en "ejecución" y archivos escritos en él.

Observaciones

- **La primera función a utilizar siempre será la que monta la memoria.**
- El contenido de los archivos no siempre está almacenado en *frames* contiguos.
- No es necesario mover los archivos para *defragmentar* las páginas y frames, es decir, se permite fragmentación interna.
- No es necesario liberar las entradas que cuenten con un bit de validación/estado, basta con establecer dichos bits en 0, lo mismo ocurre con la relación entre *frame bitmap* y *frames*.
- Si se escribe un archivo y ya no queda espacio disponible en la memoria, debe terminar la escritura. **No** debe eliminar el archivo que estaba siendo escrito.
- Cualquier detalle **no especificado** en este enunciado puede ser abarcado mediante **supuestos**, los que deben ser indicados en el `README` de su entrega.

Formalidades

La entrega se hace mediante el buzón de tareas de canvas. **Solo debe incluir el código fuente** necesario para compilar su tarea y un `Makefile`.

- La tarea debe ser realizada solamante en parejas.
- La tarea deberá ser realizada en el lenguaje de programación **C**. Cualquier tarea escrita en otro lenguaje de programación no será revisada.
- **NO debe incluir archivos binarios**[5]. En caso contrario, tendrá un descuento de 0.2 décimas en su nota final.
- **NO debe incluir un repositorio de git**[6]. En caso contrario, tendrá un descuento de 0.2 décimas en su nota final.
- Si inscribe de forma incorrecta su grupo o no lo inscribe, tendrá un descuento de 0.3 décimas
- Su tarea debe compilarse utilizando el comando `make`, y generar un ejecutable llamado `osrms` en esa misma carpeta. Si su programa **no tiene** un `Makefile`, tendrá un descuento de 0.5 décimas en su nota final y corre riesgo que su tarea no sea corregida.
- En caso de no cumplir con el formato de entrega, tendrá un descuento de 0.3 décimas en la nota final.
- Si ésta **no compila** o **no funciona** (*segmentation fault*), obtendrán la nota mínima, pudiendo recorregir modificando líneas de código con un descuento de una décima por cada cuatro líneas modificada, con un máximo de 20 lineas a modificar.

El no respeto de las formalidades o un código extremadamente desordenado podría originar descuentos adicionales, los cuales quedarán a discreción del ayudante corrector. Se recomienda modularizar, utilizar funciones y ocupar nombres de variables explicativos. En el caso de no entregar en la carpeta especificada la tarea. **no** se corregirá.

Evaluación

- **5.5 pts.** Funciones de biblioteca.
  - **0.2 pts.** `os_mount`.
  - **0.5 pts.** `os_ls_processes`.
  - **0.4 pts.** `os_exists`.
  - **0.5 pts.** `os_ls_files`.
  - **0.3 pts.** `os_frame_bitmap`.
  - **0.3 pts.** `os_tp_bitmap`.
  - **0.3 pts.** `os_start_process`.
  - **0.5 pts.** `os_finish_process`.
  - **0.4 pts.** `os_open`.
  - **0,9 pts.** `os_write_file`.
  - **0,9 pts.** `os_read_file`.
  - **0.3 pts.** `os_close`.
- **0.5 pts.** Manejo de memoria perfecto y sin errores (`valgrind`).

Importante: Se recomienda revisar el siguiente link para el detalle de la evaluación de cada item.

[5] Los archivos que resulten del proceso de compilar, como lo son los ejecutables y los *object files*

[6] Si es que lo hace, puede eliminar la carpeta oculta `.git` antes de la fecha de entrega.

Política de atraso

Se puede hacer entrega de la tarea con un máximo de 2 días hábiles de atraso. La fórmula a seguir es la siguiente:

$$N_{T_1}^{\text{Atraso}} = \text{mín}(N_{T_1}, 7{,}0 - 0{,}75 \cdot d)$$

Siendo d la cantidad de días de atraso. Notar que esto equivale a un descuento *soft*, es decir, cambia la nota máxima alcanzable y no se realiza un descuento directo sobre la nota obtenida. El uso de días de atraso no implica días extras para alguna tarea futura, por lo que deben usarse bajo su propio riesgo.

Formato de Entrega

El archivo de entrega debe ser un archivo comprimido en formato **estudiante1_estudiante2.zip**, donde **estudiante1** y **estudiante2** representan los números de alumno de los integrantes, separados por un guión bajo (_). Por ejemplo, si los números de los alumnos son **12345678** y **87654321**, el archivo deberá llamarse **12345678_87654321.zip**.

Importante: Se recomienda incluir un archivo **README.md** en el que se indiquen las funcionalidades implementadas y/o cualquier consideración que deba tenerse en cuenta al momento de corregir la tarea. Además, en caso de no cumplir con el formato de entrega, tendrá un descuento de 0.3 décimas en la nota final.

Preguntas

Cualquier duda preguntar a través del foro oficial.