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

IIC2333 — Sistemas Operativos y Redes — 1/2025

Tarea 2

Miércoles 07 de Mayo

Fecha de Entrega: Lunes 26 de Mayo a las 21:00
Composición: grupos de 2 personas

Objetivo

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

Contextualización

El proyecto consiste en desarrollar una API para escribir archivos en una memoria principal organizada en *frames*. La API deberá gestionar procesos simulados (iniciarlos, asignar memoria física, liberar memoria y eliminarlos), cada uno con su propia memoria virtual paginada. Estos procesos incluirán archivos asociados que representarán sus segmentos. Dichos archivos tendrán una dirección virtual en la memoria virtual del proceso, la cual se convertirá a direcciones físicas utilizando una tabla de páginas invertida.

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 tabla de páginas invertida.

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, 192 KB, 8 KB, 2 GB. El tamaño total de la memoria es la suma de todos estos segmentos.
- Los primeros 8 KB de la memoria corresponde a espacio reservado exclusivamente para la **Tabla de PCBs.**
- Los siguientes 192 KB están destinados a la **Tabla Invertida de Páginas.**
- Los siguientes 8 KB de la memoria corresponde a espacio reservado exclusivamente para el **Frame Bitmap.**
- Los últimos 2 GB de memoria están divididos en conjuntos de Bytes denominados *frames*:
    - **Tamaño de frame:** 32 KB. La memoria contiene un total de 2^{16} frames.

- Cada *frame* posee un número secuencial que se puede almacenar en 2 bytes, 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.
- Cantidad de entradas: 32.
- 1 Byte de estado. `0x01` si el proceso existe, o `0x00` en caso contrario.
- 14 Bytes para indicar el nombre del proceso.
- 1 Byte para indicar el `id` del proceso.
- 240 Bytes para una **Tabla de Archivos**.

Tabla de Archivos. Contiene la información de los archivos presentes en la memoria virtual de un proceso. Está separada por 10 entradas, donde cada entrada cumple con el siguiente formato:

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

Tabla Invertida de Páginas. Contiene la información para traducir direcciones virtuales a físicas dentro de la memoria. Las direcciones físicas que se obtengan serán relativas al último 2 GB de la memoria, es decir, si la dirección física relativa es dir entonces la absoluta será $2^{13} + 3 * 2^{16} + 2^{13} + dir$[2]. Finalmente, la tabla de páginas está separada en entradas las cuales tienen la siguiente estructura:

- Tamaño de entrada: 24 bits.
- Cantidad de entradas: 65536. Cada una asociada a un `PFN` de manera secuencial.
- Primer bit de cada entrada es el bit de validez. 1 indica si es válida, y 0 en otro caso.
- 10 bits para el id del proceso. Nótese que los primeros dos bits son no significativos.
- 13 bits para `VPN`. Nótese que el primer bit es no significativo.

Es importante que el conjunto de 3 Bytes sea leído como un solo tipo de dato, y luego extraer los bits. 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. Buscar en la ***Tabla Invertida de Páginas*** la entrada que presente el par con el id del proceso y la `VPN`. El índice de la entrada corresponderá al `PFN` vinculado a la página.
3. Por último, la dirección física sera igual al `PFN` seguido del `offset`.

---

1 Ese número no se encuentra guardado en la memoria física.

2 |tabla de PCBs| + |tabla de Páginas Invertida| + |Frame Bitmap| + dir .

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 está 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 *endianness* 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 *swappear* los Bytes, ya que en general los computadores trabajan con **little endian**.

API de `osrms`

Para gestionar los archivos de los procesos, tanto en operaciones de escritura como de lectura, se debe crear una biblioteca con las funciones necesarias para interactuar con la memoria principal. La implementación de la biblioteca de `osrms` se desarrollará en un archivo llamado `osrms_API.c`, y su interfaz (declaración de prototipos) estará en un archivo denominado `osrms_API.h`. Además, en `osrms_API.c` se definirá un `struct` nombrado `osrmsFile`[3], que representará un *archivo abierto* y se empleará para administrar los archivos asociados a cada proceso.

Para verificar la implementación, se debe crear un archivo (por ejemplo, `main.c`) con una función `main` que incluya el `header osrms_API.h` y utilice las funciones de la biblioteca para operar sobre un archivo que represente la memoria principal, recibido a través de la línea de comandos. La biblioteca debe incluir 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.
- `void os_ls_processes()`. Función que muestra en pantalla los procesos en ejecución en la memoria. Por cada proceso se muestra:

```
<PROCESS_ID> <PROCESS_NAME>
```

- `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`. El VPN y la dirección virtual deben estar en formato hexadecimal. Por cada archivo se muestra:

```
<VPN> <FILE_SIZE> <DIRECCION_VIRTUAL> <FILE_NAME>
```

- `void os_frame_bitmap()`. Imprime el estado actual del *Frame Bitmap*. También debe mostrar la cantidad de frames usados y libres de la siguiente manera:

[3] Use `typedef` para renombrar la estructura

```
USADOS <N_USADOS>
LIBRES <N_LIBRES>
```

.

Funciones para procesos

- `int 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. El retorno debe ser 0 si el proceso se pudo iniciar exitosamente, y -1 si algo salió mal.
- `int 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**. El retorno debe ser 0 si el poceso fue terminado exitosamente, y-1 en caso contrario.
- `int os_rename_process(int process_id, char* new_name)`. Función que cambia el nombre del proceso con identificador `process_id` al nuevo nombre dado por `new_name`. El nombre debe tener un máximo de 14 bytes. Esta función modifica directamente la entrada correspondiente en la **Tabla de PCBs** dentro de la memoria montada.

  Retorna `0` si el cambio fue exitoso, o `-1` si el proceso no existe, si el nuevo nombre es inválido, o si ocurre algún error durante la modificación.

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 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 contenido 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_delete_file(int process_id, char* file_name)`. Función para liberar memoria de un archivo perteneciente al proceso con id `process_id`. Para esto el archivo debe dejar de aparecer en la memoria virtual del proceso, además, si los *frames* quedan totalmente libres se debe indicar en el *frame bitmap* que ese *frame* ya no está siendo utilizado e invalidar la entrada correspondiente en la **Tabla Invertida de Páginas**.
- `void os_close(osrmsFile* file_desc)`. Función para cerrar archivo. Cierra el archivo indicado por `file_desc`.

Funciones bonus

- `int os_cp(int pid_src, char* fname_src, int pid_dst, char* fname_dst)`. Función que permite copiar un archivo desde un proceso origen hacia un proceso destino. La función debe abrir el archivo `fname_src` perteneciente al proceso `pid_src`, leer su contenido y crear una copia exacta en el proceso `pid_dst` bajo el nombre `fname_dst`.

  Esta operación se debe realizar utilizando las funciones existentes `os_open`, `os_read_file` y `os_write_file`, gestionando correctamente los errores que puedan surgir en cada etapa (por ejemplo, archivo no existente en el origen, espacio insuficiente en destino, etc). La función debe retornar `0` en caso de éxito, o `-1` si ocurre algún error.

  Esta función representa un desafío adicional ya que requiere coordinar múltiples operaciones de acceso y escritura sobre la memoria, por lo cual su implementación será considerada como **bonus**.

Ejecución

Para probar su biblioteca, debe usar un programa `main.c` que reciba una memoria virtual (ej: `mem.bin`). El programa `main.c` deberá usar las funciones de la biblioteca `osms_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:

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

Por otra parte, un ejemplo de una secuencia de instrucciones que se podría encontrar en `main.c` es el siguiente:

```
os_mount(argv[1]); // Se monta la memoria.
os_start_process(10, "ventas"); // Se crea el proceso 10
os_start_process(20, "reportes"); // Se crea el proceso 20
os_ls_processes(); // Lista procesos existentes
osrmsFile* f = os_open(10, "log.txt", 'w'); // Crea un archivo en el proceso 10
os_write_file(f, "log_local.txt"); // Escribe desde un archivo local
os_close(f); // Cierra el archivo
f = os_open(10, "log.txt", 'r'); // Reabre el archivo para leer
os_read_file(f, "log_copia.txt"); // Copia a archivo local
os_close(f); // Cierra el archivo
os_ls_files(10); // Lista archivos del proceso 10
os_exists(10, "log.txt"); // Verifica existencia del archivo
os_cp(10, "log.txt", 20, "reporte.txt"); // Copia el archivo a otro proceso
os_frame_bitmap(); // Imprime estado del frame bitmap
os_delete_file(10, "log.txt"); // Elimina el archivo original
os_finish_process(10); // Termina el proceso 10
os_finish_process(20); // Termina el proceso 20
```

Una vez completada la ejecución de todas las instrucciones, el archivo de memoria virtual `mem.bin` debe mostrar todos los cambios realizados. Para implementarlo, puede procesar todas las instrucciones dentro de las estructuras definidas en su programa y luego guardar el resultado final en la memoria, o alternativamente, aplicar cada instrucción directamente en `mem.bin` de manera inmediata. Lo esencial es que el estado final de la memoria virtual sea coherente con la secuencia de instrucciones ejecutada.

Para probar las funciones de su API, se hará entrega de dos memorias, las cuales serán subidas en Canvas:

- `memformat.bin`: Memoria formateada. No posee procesos ni archivos.
- `memfilled.bin`: Memoria virtual con procesos 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**[4]. En caso contrario, tendrá un descuento de 0.2 décimas en su nota final.
- **NO debe incluir un repositorio de git**[5]. 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.

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

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

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

- **6.0 pts.** Funciones de biblioteca.
  - **0.2 pts.** `os_mount`.
  - **0.6 pts.** `os_ls_processes`.
  - **0.6 pts.** `os_exists`.
  - **0.6 pts.** `os_ls_files`.
  - **0.6 pts.** `os_frame_bitmap`.
  - **0.3 pts.** `os_start_process`.
  - **0.3 pts.** `os_finish_process`.
  - **0.2 pts.** `os_rename_process`.
  - **0.3 pts.** `os_open`.
  - **0.3 pts.** `os_close`.
  - **0.6 pts.** `os_write_file`.
  - **0.6 pts.** `os_read_file`.
  - **0.8 pts.** `os_delete_file`.
- **1.0 pt (Bonus)** `os_cp`.
- **Descuento: Manejo de Memoria** Se descontará hasta **1.0 pts** si `valgrind` no reporta en su código 0 *leaks* y 0 errores de memoria en **todo caso de uso**.

Corrección

A diferencia de las tareas, este proyecto será corregido junto a ustedes. Se hará uso de la plataforma Google Meets para llevarlo a cabo. Esto se hará de la siguiente forma:

1. Como pareja, deberán elaborar uno o más *scripts* `main.c` que hagan uso de **todas las funciones de la API que hayan implementado de forma correcta**. Si implementan una función en su librería pero no evidencian su funcionamiento en la corrección, **no será evaluada**.
2. Los *scripts* `main.c` deben ser subidos a Canvas en la fecha de entrega como también todos los archivos que estimen necesarios para la corrección.
3. Para que el proceso sea transparente, se descargarán desde el servidor los *scripts* de su API y, en conjunto con el `main.c` elaborado, le mostrarán a los ayudantes el funcionamiento de su programa.
4. Pueden usar los archivos que deseen y de la extensión que deseen para evidenciar el funcionamiento correcto de su API. Como recomendación, los archivos `gif` y de audio son muy útiles para mostrar las limitantes del tamaño de los archivos.
5. Puede (y se recomienda) hacer uso de más de un programa `main.c`, de forma que estos evidencien distintas funcionalidades de su API.

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.

Preguntas

Cualquier duda, a través del foro oficial. 48 horas antes de la hora de entrega se dejarán de responder dudas.