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

IIC2333 — Sistemas Operativos y Redes — 1/2021

Proyecto 1

Viernes 24 de Septiembre, 2021

Fecha de Entrega: Lunes 11 de Octubre, 2021
Composición: grupos de 5 personas
Fecha de ayudantía: Grabada, 2021

Objetivo

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

Contextualización

El proyecto consistirá en crear una API que escriba archivos en una memoria principal dividida en *frames*. Se escribirán archivos en esta memoria y se espera leer de manera consistente sus contenidos. La API deberá manejar procesos simulados (ejecutarlos, asignar memoria física, liberar memoria, y terminarlos), donde cada uno de estos procesos tendrá su propia memoria virtual paginada y una tabla de páginas. Respecto a los archivos, cada uno de estos estará asociado a un proceso, por lo que cuando se escriben deberán tener una dirección dentro de la memoria virtual del proceso correspondiente, y será deber de los alumnos transformar, con ayuda de la tabla de páginas, dichas dirección virtuales a direcciones dentro de la memoria física.

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*. Tanto páginas como *frames* deben ser del mismo tamaño.

En este proyecto tendrán la posibilidad de experimentar con una implementación de un mecanismo de paginación simplificado 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** y **paginación**.

Estructura de memoria `crms`

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

- Tamaño de memoria física: 4KB + 16 B + 1 GB.
- Los primeros 4KB de la memoria corresponde a espacio reservado exclusivamente para la **Tabla de PCBs**.
- Luego, existen 16 Bytes destinados al **frame bitmap**.
- El siguiente 1 GB de memoria está dividido en conjuntos de Bytes denominados *frames*:
    - Tamaño de *frame*: 8 MB. La memoria contiene un total de 128 *frames*.

- Cada *frame* posee un numero secuencial que se puede almacenar en 7 bits, por lo cual se puede guardar en un `unsigned int`. **Este valor corresponde a su PFN**[1].

Cada proceso cuenta con un espacio virtual de memoria igual a 256 MB. Esta memoria virtual está dividida en páginas de 8 MB.

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

- Tamaño de entrada: 256 Bytes.
- Cantidad de entradas: 16.
- 1 Byte de estado. `0x01` si el proceso está en ejecución, o `0x00` en caso contrario.
- 1 Byte para indicar el `id` del proceso.
- 12 Bytes para indicar el nombre del proceso.
- Luego de los primeros 14 Bytes, cada entrada tiene 10 subentradas para guardar información sobre los archivos de su memoria virtual.Cada una de estas subentradas contiene:
  - 1 Byte de validez. `0x01` si la entrada es válida, o `0x00` en caso contrario.
  - 12 Bytes para nombre del archivo.
  - 4 Bytes para tamaño del archivo. El tamaño máximo de un archivo es de 32 MB.
  - 4 Bytes para la dirección virtual. 4 bits no significativos `(0b0000)` + 5 bits VPN + 23 bits offset.
- Los últimos 32 Bytes corresponderán a **tabla de páginas** del proceso.

Tabla 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 1 GB de la memoria, es decir, si la dirección física relativa es dir entonces la absoluta será $2^{12} + 2^4 + dir$ [2]. Finalmente, la tabla de páginas está separada en entradas las cuales tienen la siguiente estructura:

- Tamaño de entrada: 1 Byte. `0x01` si es válida, o `0x00` en caso contrario.
- Cantidad de entradas: 32
- Primer bit de cada entrada es el bit de validez. `1` indica que la entrada es válida, y `0` indica que es inválida.
- 7 bits para `PFN`.

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

1. Obtener de la dirección virtual los 5 bits del `VPN` y 23 bits del `offset`.
2. Ingresar a la entrada `VPN` de la tabla de páginas y obtener el `PFN`.
3. Luego, la dirección física será igual a `PFN` seguido del `offset`.

Existen 32 entradas en tabla de páginas, una para cada página. Cada entrada está asociada a un `VPN` de manera secuencial. Por ejemplo, si `VPN` es igual a 0 entonces su entrada correspondiente será la primera de la tabla de páginas.

Frame bitmap. Se encuentra a continuación de la tabla de PCBs y cuenta con un tamaño de 16 Byte = 128 bit. 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 conclusión:

---

1 Ese número no se encuentra guardado en la memoria física

2 En otras palabras, |tabla de PCBs| + |Frame bitmap| + dir

- 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.

Frame. Aquí se almacenan los datos de los archivos. El tamaño de cada *frame* es de 8MB. Se encuentran en el último 1 GB y en total hay 128 *frames*.

Es importante destacar que la lectura y escritura de Byte en los bloques de datos **deben** ser realizadas en orden ***big endian***, para mantener consistencia entre todos los sistemas de archivos implementados.

API de `crms`

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 `crms_API.c` y su interfaz (declaración de prototipos) debe encontrarse en un archivo de nombre `crms_API.h`.
Para probar su implementación debe escribir un archivo (por ejemplo, `main.c`) con una función `main` que incluya el *header* `crms_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. Dentro de `crms_API.c` se debe definir un `struct` que almacene la información que considere necesaria para operar con el archivo. Ese `struct` debe ser nombrado `CrmsFile` mediante una instrucción `typedef`. Esta estructura representará un *archivo abierto* y será utilizada para manejar los archivos pertenecientes a cada proceso. .

La biblioteca debe implementar las siguientes funciones.

Funciones generales

- `void cr_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 cr_ls_processes()`. Función que muestra en pantalla los procesos en ejecución.
- `int cr_exists(int process_id, char* file_name)`. Función para ver 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 cr_ls_files(int process_id)`. Función para listar los archivos dentro de la memoria del proceso. Imprime en pantalla los nombres de todos los archivos presentes en la memoria del proceso con id `process_id`.

Funciones procesos

- `void cr_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**.
- `void cr_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**.

Funciones archivos

- `CrmsFile* cr_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 `CrmsFile*` que lo representa. Si `mode` es `'w'`, se verifica que el archivo no exista y se retorna un nuevo `CrmsFile*` que lo representa.

- `int cr_write_file(CrmsFile* file_desc, void* buffer, int n_bytes)`. Función para escribir archivos. Escribe en el archivo descrito por `file_desc` los `n_bytes` que se encuentren en la dirección indicada por `buffer` y retorna la cantidad de Bytes escritos en el archivo[3]. Cabe recalcar que los archivos parten con tamaño 0 y luego su tamaño crece a medida que se escribe. 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 contiguo en la memoria virtual, ó
  - Se escribieron los `n_bytes`.
- `int cr_read(CrmsFile* file_desc, void* buffer, int n_bytes)`. Función para leer archivos. Lee los siguientes `n_bytes` desde el archivo descrito por `file_desc` y los guarda en la dirección apuntada por `buffer`. Debe retornar la cantidad de Bytes efectivamente leídos desde el archivo[4]. La lectura de `cr_read` se efectúa recorriendo los *frames* donde se encuentra escrito su contenido, comenzando desde la última posición leída por un llamado a `cr_read`.

  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 completar los `n_bytes` o llegar al final del archivo.
- `void cr_delete_file(CrmsFile* file_desc)`. 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 de páginas.
- `void cr_close(CrmsFile* 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 una memoria virtual (ej: `mem.bin`). El programa `main.c` deberá usar las funciones de la biblioteca `crms_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:

```
./crms mem.bin
```

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

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

Al terminar de ejecutar todas las instrucciones, la memoria virtual `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

[3] Esto es importante cuando no se logra escribir los `n_bytes`
[4] Esto es importante si `n_bytes` es mayor a la cantidad de Bytes restantes en el archivo

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. Se podrá descargar del servidor a través de la siguiente ruta:

  `/home/iic2333/P1/memformat.bin`

- `memfilled.bin`: Memoria virtual con procesos en "ejecución" y archivos escritos en él. Se podrá descargar del servidor a través de la siguiente ruta:

  `/home/iic2333/P1/memfilled.bin`

Bonus

En este proyecto habrá un bonus de 10 décimas que consiste en implementar **manejo de errores estilo `errno`**. La librería `errno` funciona definiendo una variable global en la cual las funciones guardan un código de error en caso de que algo haya pasado.

Para obtener el bonus el proyecto deberá cumplir con las siguientes condiciones:

- Definición de la variable global `CR_ERROR`.
- Definición de un enum[5] que contenga todos los códigos de error. Estos deben ser lo más genéricos[6] posible.
- Definición de una función `void cr_strerror` que reciba un código de error e imprima en consola un mensaje que lo explique de forma breve.
- Cada función de la API debe guardar un código de error relevante en `CR_ERROR` en caso de error.
- Escribir un breve informe que explique sus códigos de error, en que casos aplican y cuál es su mensaje asociado.

Se deben cumplir, al menos parcialmente, con todos estos puntos. La cantidad de décimas de bonus que se obtenga depende de:

- Cuanta cobertura de errores se logró.
- Que tan genéricos son los códigos de error.
- Que tan útiles son sus mensajes de errores.
- Proyectos con nota mayor o igual a $4{,}0$ pueden obtener hasta 5 décimas de bonus.
- Proyectos a nota mayor o igual a $5{,}0$ pueden obtener hasta 10 décimas de bonus.

Corrección "presencial"

A diferencia de las tareas, este proyecto será corregido de **forma "presencial"**. Se hará uso de la plataforma Google Meets para llevarla a cabo. Esto se hará de la siguiente forma:

---

5 Pueden leer de enums en el siguiente enlace.

6 Por ejemplo, en vez de tener códigos de error que validen los *input* de cada función, se puede tener uno solo que englobe ese caso.

1. Como grupo, 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**. Es importante manejar bien los tiempos para que alcancen a mostrar todo lo que implementaron.
2. No es necesario que los *scripts* `main.c` sean subidos al servidor en la fecha de entrega, pero sí que los compartan al momento de llevar a cabo 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
5. Puede (y se recomienda) hacer uso de más de un programa `main.c`, de forma que estos evidencien distintas funcionalidades de su API.

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 específicado** en este enunciado puede ser abarcado mediante **supuestos**, los que deben ser indicados en el `README` de su entrega.

Formalidades

Deberá incluir un `README` que indique quiénes son los autores del proyecto (**con sus respectivos números de alumno**), cuáles fueron las principales decisiones de diseño para construir el programa, qué supuestos adicionales ocuparon, y cualquier información que considere necesaria para facilitar la corrección. Se sugiere utilizar formato **Markdown**.

La tarea **debe** ser programada en **C**. No se aceptarán desarrollos en otros lenguajes de programación.

La entrega del código de su API será a traves del servidor del curso, en la fecha estipulada. La entrega del proyecto deberá ser en una carpeta llamada `P1`, en la carpeta de cualquiera de los integrantes del curso.

El no respeto de las formalidades o un código extremadamente desordenado podría originar descuentos adicionales, los que son detallados en la sección correspondiente. Se recomienda modularizar, utilizar funciones y ocupar nombres de variables explicativos. En el caso de no entregar en la carpeta especificada, el proyecto **no se corregirá**.

Evaluación

- **1.2 pts.** Estructura de memoria.
  - **0.4 pts.** Representación **tabla de PCBs**.
  - **0.4 pts.** Representación **tabla de páginas**.
  - **0.2 pts.** Representación de **frame bitmap**.
  - **0.2 pts.** Representación de **frame**.
- **3.8 pts.** Funciones de biblioteca.
  - **0.1 pts.** `cr_mount`.
  - **0.3 pts.** `cr_ls_processes`.
  - **0.2 pts.** `cr_exists`.
  - **0.3 pts.** `cr_ls_files`.
  - **0.3 pts.** `cr_start_process`.
  - **0.3 pts.** `cr_finish_process`.
  - **0.4 pts.** `cr_open`.
  - **0.6 pts.** `cr_write_file`.
  - **0.6 pts.** `cr_read_file`.
  - **0.6 pts.** `cr_delete_file`.
  - **0.1 pts.** `cr_close`.
- **1.0 pts.** Manejo de memoria perfecto y sin errores (`valgrind`).

Descuentos

- Descuento de 0.5 puntos por subir **archivos binarios** (programas compilados).
- Descuento hasta de 1 punto por no seguir las formalidades.
- Descuento de 1 punto si la entrega **no tiene un** `Makefile`**, no compila o no funciona (*segmentation fault*)**.
- Descuento de 3 puntos si se sube alguno de los archivos `memoria.bin` al servidor.