AYUDANTÍA P1-2

Germán Leandro Contreras Sagredo
Ricardo Esteban Schilling Broussaingaray

IIC2333 [2019-2] - Sistemas Operativos y Redes

INTRODUCCIÓN

INTRODUCCIÓN

Los objetivos de esta ayudantía son:

1. Responder dudas sobre la estructura de nuestro sistema de archivos y del funcionamiento esperado de la API `crfs`.
2. Aprender a leer segmentos de archivos binarios desde un puntero e interpretar lo leído según lo que indique la estructura del sistema.
3. Ver ejemplos en código y realizar un ejemplo práctico para avanzar en el proyecto.

LECTURA DE ARCHIVOS BINARIOS

LECTURA DE ARCHIVOS BINARIOS

- Para poder leer el archivo binario, sí o sí necesitarán hacer uso de la estructura `FILE`. Como los métodos no reciben la ruta de este, debe quedar establecida en una **variable global**, de forma que se pueda acceder a esta desde cualquier función de la API.
- Luego, tienen que hacer uso de las funciones de lectura y escritura para leer y modificar el disco. No obstante, también deben ser capaces de posicionar el puntero para poder leer el bloque que desean.
- **NO DEBEN LEER EL ARCHIVO BINARIO Y DEJARLO EN MEMORIA.** Hará que su implementación sea muy lenta, lo que dificultará de sobremanera su uso.

LECTURA DE ARCHIVOS BINARIOS

- A continuación, veremos la descripción de las funciones que deberán usar para llevar a cabo su proyecto. Deben incluir la librería `stdio.h`.
- Pueden o no usarlas todas, pero se recomienda revisarlas para poder facilitar algunas de las tareas solicitadas.
- Después de la descripción textual, se mostrará un ejemplo de código que muestre el uso correcto de las funciones.

LECTURA DE ARCHIVOS BINARIOS – FUNCIONES

- `fopen:` Permite abrir un archivo, en modo lectura o escritura. Retorna una estructura `FILE`.
- `fclose:` Cierra un archivo previamente abierto. **Asegúrense de hacerlo siempre.**
- `fseek:` La función más importante para seguir la estructura del sistema de archivos. Posiciona el puntero "lector" de un archivo según dos argumentos: `offset` (la cantidad de bytes a moverse) y `whence` (la posición desde la cual se mueve la cantidad de bytes ingresada). Esta última tiene tres valores posibles:
  - `SEEK_SET:` Desde el comienzo del archivo (posición 0). Se puede indicar con un 0.
  - `SEEK_CUR:` Desde la posición actual del puntero -lo que dependerá de si hemos aplicado `fseek` o `fread` antes o no-. Se puede indicar con un 1.
  - `SEEK_END:` Desde el final del archivo -útil solo si se quiere escribir algo al final-. Se puede indicar con un 2.

LECTURA DE ARCHIVOS BINARIOS – FUNCIONES

- `fread:` Permite leer una cantidad definida de bytes de un archivo y las almacena en un puntero `buffer`. Se debe indicar el tamaño de cada elemento a leer y la cantidad de elementos. En nuestro caso, lo más simple es indicar un tamaño de 1 byte e indicar la cantidad de bytes a leer.
- `fwrite:` Permite escribir una cantidad definida de bytes en un archivo desde un puntero `buffer`. Se debe indicar el tamaño de cada elemento a escribir y la cantidad de elementos.

En ambos casos, el puntero del archivo se mueve en una cantidad igual a los bytes escritos o leídos... ¿A qué se parece esto? cr_read, cr_write.

Si en vez de una cantidad de bytes, desean solo leer/escribir uno... fgetc, fputc permiten leer y escribir un caracter en un archivo respectivamente... pero es importante recordar que **los caracteres son de un byte cada uno**.

LECTURA DE ARCHIVOS BINARIOS – FUNCIONES

- `ftell:` Retorna la posición del puntero del archivo. Asumiendo que se mide el tamaño en bytes, si insertamos el puntero al final, nos retorna el número de bytes del archivo. ¿Nos sirve esto para algo? Pensemos en las funciones cr_load, que carga un archivo de la máquina local al disco binario. Esta no recibe como argumento el número de bytes del archivo a cargar... aquí cobra importancia el uso de esta función.

LECTURA DE ARCHIVOS BINARIOS – VOID*

- Habrán notado que muchas funciones de la API en el proyecto hacen uso de void*. Esto se utiliza cuando queremos tener un puntero de datos **sin estructura fija**.
- Esto es muy conveniente para este contexto, dado que lo que leemos **no siempre es de un único tipo de dato**. ¿Qué tipos de dato podemos encontrar si pensamos **solo** en el bloque índice de un archivo?
- Entonces, a la función `cr_read` le podemos pasar un puntero de cualquier tipo de dato para después darle una interpretación según lo que se esté leyendo. En general se recomienda trabajar con unsigned char*, ya que cada caracter se puede traducir en un byte y es el tipo de dato más recurrente, aunque también será necesario trabajar con unsigned int para algunos casos.

LECTURA DE ARCHIVOS BINARIOS – EJEMPLO

```c
/* Haremos uso de fseek para ver la cantidad de bytes
de un archivo y leerlo */
FILE* f = fopen(path_1, "rb"); // Lectura binaria.
fseek(f, 0, SEEK_END); // Ponemos el puntero al final.
int size_f = ftell(f); // Retorna la posición del puntero.
fseek(f, 0, SEEK_SET); // Lo devolvemos al inicio.
unsigned char buffer[size_f]; // Buffer del tamaño de
                              // bytes de f.
fread(buffer, 1, size_f, f); // Leemos los bytes y los
                             // almacenamos en el buffer.
fclose(f); // Cerramos el archivo.
/* Haremos lo mismo, pero para escribir un nuevo archivo. */
FILE* f_new = fopen(path_2, "wb"); // Escritura binaria.
fwrite(buffer, 1, size_f, f_new);  // Escribimos la misma
                                   // cantidad de bytes
                                   // desde el buffer.
fclose(f_new); // Cerramos el archivo.
```

LECTURA DE ARCHIVOS BINARIOS – OPERACIONES BITWISE

- & Bitwise AND, el bit resultante es 1 ssi ambos bits del input son 1.
- | Bitwise OR, el bit resultante es 1 si uno o ambos bits del input son 1.
- $\wedge$ Bitwise XOR, el bit resultante es 1 ssi uno de los bits del input es uno, pero no ambos.
- $\sim$ Bitwise NOT, invierte los bits del input.
- « Left Shift, mueve hacia la izquierda los bits del primer input la cantidad de veces que lo indique el segundo input.
- » Right shift, lo mismo que el anterior pero mueve los bits a la izquierda.

En C, los shifts a los números de tipo `unsigned` son shifts lógicos, es decir que llenan los espacios "nuevos" con ceros, mientras que hacer shift a un número `signed` realizará un shift aritmético, llenando el nuevo espacio con el bit eliminado (recuerdo de `Assembly`: `rotate`).

LECTURA DE ARCHIVOS BINARIOS – OPERACIONES BITWISE

- ¿Para qué nos puede servir esto?
- Operar con los bits individuales de un byte, por ejemplo, para buscar en el bloque de bitmap algún bloque vacío.
- Obtener un bit específico que nos interesa, lo que nos puede servir para identificar si una entrada de directorio es válida, por ejemplo.

LECTURA DE ARCHIVOS BINARIOS – EJEMPLOS BITWISE

```c
unsigned int a = 60;  /* 60 = 0011 1100 */
unsigned int b = 13;  /* 13 = 0000 1101 */
int c = 13; /* 13 = 0000 1101 */
```

- `a & b = 12 /* 0000 1100 */`
- `a | b = 61 /* 0011 1101 */`
- `a ^ b = 49 /* 0011 0001 */`
- `~a = 195 /* 1100 0011 */`
  `~b = 242 /* 1111 0010 */`
- `b << 1 = 26 /* 0001 1010 */`
  `b << 2 = 52 /* 0011 0100 */`
- `b >> 1 = 6 /* 0000 0110 */`
  `b >> 2 = 3 /* 0000 0011 */`
  `c >> 1 = -122 /* 1000 0110 */`
  `c >> 2 = 67 /* 0100 0011 */`

TIPS

TIPS - TRADUCCIÓN BYTES-ENTEROS

```c
unsigned char pointer_bytes[4];
unsigned int pointer = 1234567;
/* Entero -> Bytes */
pointer_bytes[3] = pointer; /* El byte menos significativo */
pointer_bytes[2] = (pointer >> 8); /* SHRx8 -> 2° byte ahora es el 1° */
pointer_bytes[1] = (pointer >> 16); /* SHRx16 -> 3° byte ahora es el 1° */
pointer_bytes[0] = (pointer >> 24); /* SHRx24 -> 4° byte ahora es el 1° */
/* Bytes -> Entero */
/* Se hace uso de SHL para obtener el valor original de cada byte */
unsigned int new_pointer = pointer_bytes[3] + (pointer_bytes[2] << 8)
                 + (pointer_bytes[1] << 16) + (pointer_bytes[0] << 24);
printf("Byte 0: %i\nByte 0 << 24: %i\n", pointer_bytes[0],
       pointer_bytes[0] << 24);
printf("Byte 1: %i\nByte 1 << 16: %i\n", pointer_bytes[1],
       pointer_bytes[1] << 16);
printf("Byte 2: %i\nByte 2 << 8: %i\n", pointer_bytes[2],
       pointer_bytes[2] << 8);
printf("Byte 3: %i\n", pointer_bytes[3]);
printf("Pointer value: %i", new_pointer);
```

TIPS - MANEJO INDIVIDUAL DE BITS

```c
unsigned int x = 250;
/* ¿Cómo saber que bit es igual a 1 o 0? */
printf("Bit 7: %i\n", (x >> 7) % 2);
printf("Bit 6: %i\n", (x >> 6) % 2);
printf("Bit 5: %i\n", (x >> 5) % 2);
printf("Bit 4: %i\n", (x >> 4) % 2);
printf("Bit 3: %i\n", (x >> 3) % 2);
printf("Bit 2: %i\n", (x >> 2) % 2);
printf("Bit 1: %i\n", (x >> 1) % 2);
printf("Bit 0: %i\n", x % 2);
/* X mod 2 = bit menos significativo (par-impar). */
printf("========\n");
/* ¿Otra forma? */
printf("Bit 7: %i\n", (x >> 7) & 1);
printf("Bit 6: %i\n", (x >> 6) & 1);
printf("Bit 5: %i\n", (x >> 5) & 1);
printf("Bit 4: %i\n", (x >> 4) & 1);
printf("Bit 3: %i\n", (x >> 3) & 1);
printf("Bit 2: %i\n", (x >> 2) & 1);
printf("Bit 1: %i\n", (x >> 1) & 1);
printf("Bit 0: %i", x & 1);
/* Operación & 1 es una máscara: X AND 00000001 */
/* Solo quedará el valor del bit menos significativo. */
```

ENLACES Y RECURSOS ÚTILES

ENLACES Y RECURSOS ÚTILES

- TutorialsPoint – Funciones de manejo de archivos
- TutorialsPoint – Operaciones bitwise

FIN