Recuperatorio Primer Parcial

Primer Cuatrimestre 2024

Normas generales

- El parcial es INDIVIDUAL
- Una vez terminada la evaluación se deberá completar un formulario con el *hash* del *commit* del repositorio de entrega. El link al mismo es: `https://forms.gle/pTwNZLtatMi6YMm9A`.
- De ser necesario, luego de la entrega habrá una instancia coloquial de defensa del trabajo

Régimen de Aprobación

- Para aprobar el examen es necesario obtener como mínimo **60 puntos**.

Compilación y Testeo

Para compilar y ejecutar los tests se dispone de un archivo `Makefile` con los siguientes *targets*

`make test_c`	Genera el ejecutable usando la implementación en C del ejercicio.
`make test_asm`	Genera el ejecutable usando la implementación en ASM del ejercicio.
`make run_c`	Corre los tests usando la implementación en C.
`make run_asm`	Corre los tests usando la implementación en ASM.
`make valgrind_c`	Corre los tests en valgrind usando la implementación en C.
`make valgrind_asm`	Corre los tests en valgrind usando la implementación en ASM.

Además, tener en cuenta lo siguiente:

- Cada ejercicio tiene su propio `Makefile` con los targets arriba mencionados
- Es importante que lean la documentación provista en el archivo header correspondiente (`ej1.h` o `ej2.h` según corresponda). La documentación completa de las funciones se encuentra en dichos headers, no en el presente enunciado.
- Todos los incisos del parcial son independientes entre sí.
- El sistema de tests del parcial sólo correrá los tests que hayan marcado como hechos. Para esto deben modificar la variable `EJERCICIO_xx_HECHO` correspondiente asignándole `true` (en C) ó `TRUE` (en ASM). `xx` es el inciso en cuestión: `1A`, `1B`, `1C`, `2A` o `2B`.

Ej. 1 - (50 puntos)

En este ejercicio vamos a explorar un *framework* sencillo para manejar cadenas de caracteres el cual nos va a permitir combinar en forma arbitraria cadenas simples para formar cadenas de caracteres más complejas.

Las cadenas se representan haciendo uso de tres tipos `texto_literal_t`, `texto_concatenacion_t` y `texto_cualquiera_t`.
El primero, `texto_literal_t` representa un string sencillo:

```
typedef struct {
        uint32_t tipo;
        uint32_t usos;
        uint64_t tamanio;
        const char* contenido;
} texto_literal_t;
```

`tipo`	Siempre es el valor numérico `TEXTO_LITERAL` (Ver defines en `ej1.h`).
`usos`	La cantidad de otras cadenas que usan esta cadena. Al momento de su creación este valor es 0.
`tamanio`	La cantidad de caracteres de la cadena (sin contar el carácter nulo de terminación).
`contenido`	La cadena en sí (representada como una *C string* clásica).

El segundo, `texto_concatenacion_t` representa una concatenación de dos cadenas:

```
typedef struct {
        uint32_t tipo;
        uint32_t usos;
        texto_cualquiera_t* izquierda;
        texto_cualquiera_t* derecha;
} texto_concatenacion_t;
```

`tipo`	Siempre es el valor numérico `TEXTO_CONCATENACION` (Ver defines en `ej1.h`).
`usos`	La cantidad de otras cadenas que usan esta cadena. Al momento de su creación este valor es 0.
`izquierda`	La cadena de la izquierda. Puede referenciar a un `texto_literal_t` o a un `texto_concatenacion_t`.
`derecha`	La cadena de la derecha. Puede referenciar a un `texto_literal_t` o a un `texto_concatenacion_t`.

Nótese que el tipo `texto_concatenacion_t` **no contiene las cadenas en sí**, sino que mantiene solo las referencias.

Finalmente, `texto_cualquiera_t` sirve para poder referirse a cualquiera de los dos tipos anteriores de manera genérica:

```
typedef struct {
        uint32_t tipo;
        uint32_t usos;
        uint64_t unused0; // Reservamos espacio
        uint64_t unused1; // Reservamos espacio
} texto_cualquiera_t;
```

A continuación algunos ejemplos gráficos de como se podrían usar estos tipos de datos. Las estructuras marcadas en rojo representan `texto_literal_t` y las marcadas en negro son `texto_concatenacion_t`

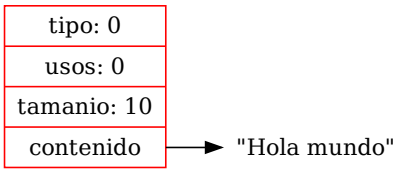

Figura 1: Ejemplo de un texto constituído por un único literal.
Representa la cadena `Hola mundo`.

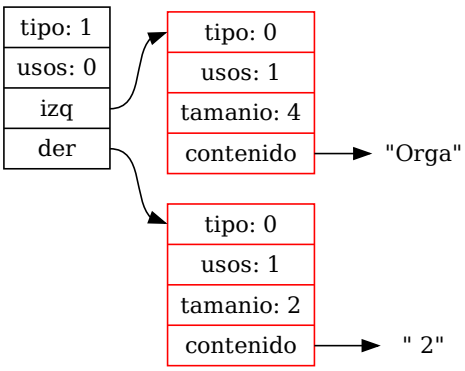

Figura 2: Ejemplo de un texto constituído por una concatenación de dos literales.
Representa la cadena `Orga 2`. Nótese el espacio en el string de la derecha.

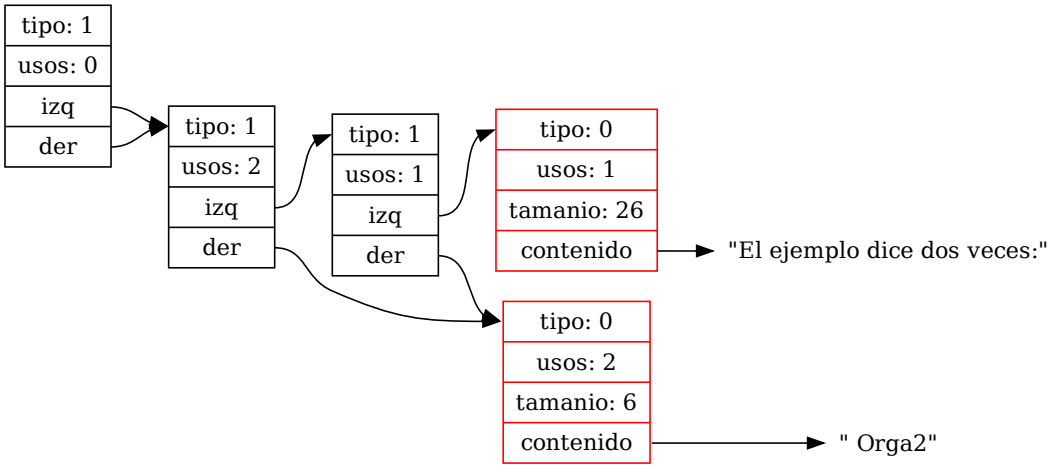

Figura 3: Ejemplo de un texto constituído por una concatenación de concatenaciones de literales.
Representa la cadena `El ejemplo dice dos veces: Orga2 Orga2 El ejemplo dice dos veces: Orga2 Orga2`.

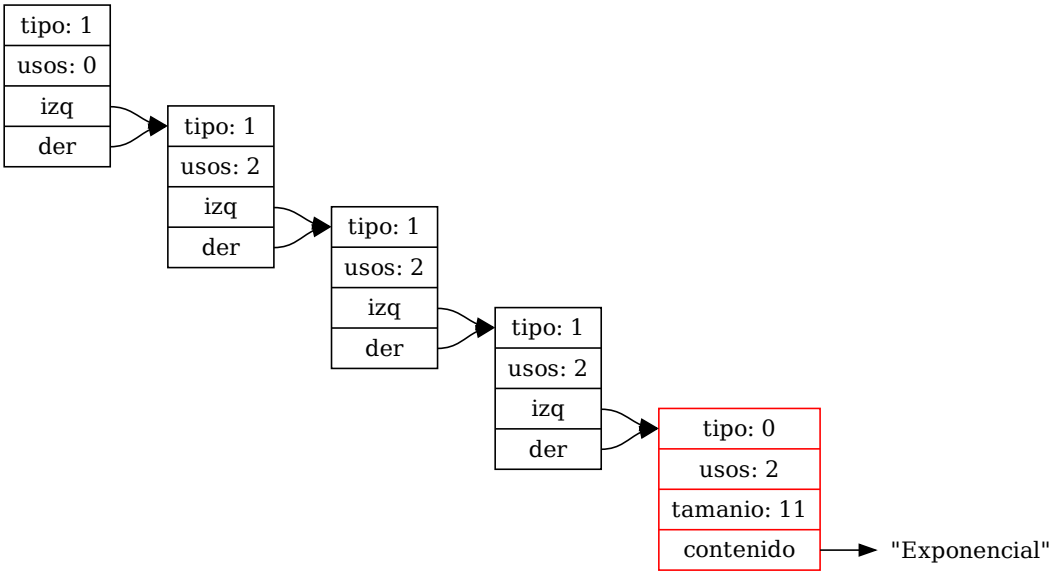

Figura 4: Ejemplo de un texto constituído por una concatenación que hace reuso de concatenaciones duplicando la longitud del texto por cada nivel.
Representa la cadena `ExponencialExponencialExponencialExponencialExponencialExponencialExpo` (16 repeticiones).

A su vez disponen de las siguientes funciones, que son parte del *framework*:

```
/* Imprime un texto */
void texto_imprimir(texto_cualquiera_t* texto);

/* Libera el texto pasado como parámetro */
bool texto_liberar(texto_cualquiera_t* texto);
```

IMPORTANTE:
Para entender qué hace cada función lean la documentación completa en `ej1.h`

A entregar:

Se pide realizar en assembly (en el archivo `ej1.asm`) las siguientes funciones:

a)
```
/* Crea un `texto_literal_t` que representa la cadena pasada por parámetro.*/
texto_literal_t* texto_literal(const char* texto);

/* Crea un `texto_concatenacion_t` que representa la concatenación de ambos parámetros.*/
texto_concatenacion_t* texto_concatenar(texto_cualquiera_t* izquierda, texto_cualquiera_t* derecha);
```

b)
```
/* Calcula el tamaño total de un `texto_cualquiera_t`. Es decir, suma todos
 * los campos `tamanio` involucrados en el mismo. */
uint64_t texto_tamanio_total(texto_cualquiera_t* texto);
```

c)
```
/* Chequea si los tamaños de todos los nodos literales internos al parámetro corresponden al
 * tamaño de la cadena apuntada.
 * (es decir: si los campos `tamanio` están bien calculados). */
bool texto_chequear_tamanio(texto_cualquiera_t* texto);
```

Tanto el archivo `ej1.c` como `ej1.asm` tienen implementaciones de `texto_imprimir` y `texto_liberar`. Para que las implementaciones en ASM funcionen correctamente **deben completar las definiciones de sizes y offsets que figuran en** `ej1.asm`.

Ej. 2 - (50 puntos)

Pese a sus exitosas ventas la *OrgaStation2* no es la consola más potente del mercado. Para obtener buena fidelidad gráfica muchos juegos decidieron pre-renderizar contenido 3D el cual mezclan con gráficos propios de la consola.

Hacer esto es relativamente fácil. Se toman dos imágenes de las cuales se tiene su mapa de color (4 canales, RGBA, 8 bits por canal, sin signo) y su mapa de profundidad (1 canal, escala de grises, 8 bits por canal, sin signo). Por cada píxel en las imágenes fuente se elije el de **menor profundidad** y se escribe en la imagen destino. Es decir:

$$\texttt{dst}[y,x] = \begin{cases} \texttt{a}[y,x] & \text{si } \texttt{depth\_a}[y,x] < \texttt{depth\_b}[y,x] \\ \texttt{b}[y,x] & \text{sino} \end{cases}$$

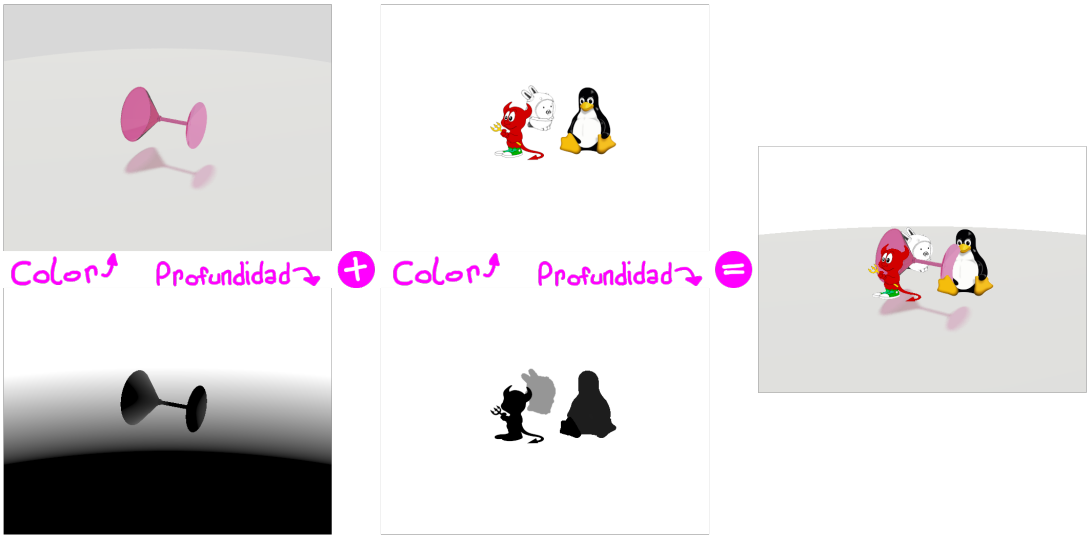

Figura 5: Una imagen compleja puede mezclarse con otra que es una serie de planos. Algunos planos quedan detrás de la imagen y otros por delante.

Figura 6: Dos escenas tridimensionales complejas pueden mezclarse. Los elementos cercanos a la cámara predominan en el resultado.

Algunos programas miden el mapa de profundidad en metros mientras que otros en pies, yardas o centímetros. Por esto, cada mapa tiene también dos factores de corrección: `scale` y `offset`. Para evitar pérdidas de precisión al aplicar esta corrección el mapa de profundidad corregido usa enteros de 32 bits con signo. Es decir, el mapa de precisión corregido es (1 canal, escala de grises, 32 bits, con signo). La forma en la que se aplica la corrección es:

$$\texttt{dst\_depth}[y, x] = \texttt{scale} \times \texttt{uint8\_to\_int32}(\texttt{src\_depth}[y, x]) + \texttt{offset}$$

A entregar:

En ASM, utlizando SIMD y teniendo como objetivo el procesamiento de 4 pixeles en simultáneo, se pide:

a) Implementar el algoritmo de corrección. El mismo realiza tanto la conversión de 8 bits sin signo a 32 bits con signo como la aplicación de la transformación lineal especificada por `scale` y `offset`. La corrección es básicamente hacer $\texttt{dst\_depth}[y, x] = \texttt{scale} \times \texttt{src\_depth}[y, x] + \texttt{offset}$ por cada píxel de `dst`.

```
void ej2a(
        int32_t* dst_depth,
        uint8_t* src_depth,
        int32_t scale, int32_t offset,
        uint32_t width, uint32_t height
);
```

Parámetros:

`dst_depth` La imagen destino (mapa de profundidad). Está en escala de grises a 32 bits con signo por canal.

`src_depth` La imagen origen (mapa de profundidad). Está en escala de grises a 8 bits sin signo por canal.

`scale` El factor de escala. Es un entero con signo de 32 bits. Multiplica a cada píxel de la entrada.

`offset` El factor de corrimiento. Es un entero con signo de 32 bits. Se suma a todos los píxeles luego de escalarlos.

`width` El ancho en píxeles de `src` y `dst`.

`height` El alto en píxeles de `src` y `dst`.

b) Implementar el algoritmo de *"mezclado sensible a la profundidad"*. De forma breve este algoritmo se puede describir como hacer $\texttt{dst}[y, x] = \begin{cases} \texttt{a}[y, x] & \text{si } \texttt{depth\_a}[y, x] < \texttt{depth\_b}[y, x] \\ \texttt{b}[y, x] & \text{sino} \end{cases}$ en cada píxel de `dst`. En caso de emapte entre las profundidades de A y B se debe escribir el píxel de B.

```
void ej2b(
        rgba_t* dst,
        rgba_t* a, int32_t* depth_a,
        rgba_t* b, int32_t* depth_b,
        uint32_t width, uint32_t height
);
```

Parámetros:

`dst` La imagen destino. Está a color (RGBA) en 8 bits sin signo por canal.

`a` La imagen origen A. Está a color (RGBA) en 8 bits sin signo por canal.

`depth_a` El mapa de profundidad de A. Está en escala de grises a 32 bits con signo por canal.

`b` La imagen origen B. Está a color (RGBA) en 8 bits sin signo por canal.

`depth_b` El mapa de profundidad de B. Está en escala de grises a 32 bits con signo por canal.

`width` El ancho en píxeles de todas las imágenes parámetro.

`height` El alto en píxeles de todas las imágenes parámetro.

Se puede asumir:

- Tanto el ancho como el alto de las imágenes (en píxeles) es múltiplo de 16
- scale $\times$ profundidad + offset es siempre representable como entero de 32 bits con signo.
- Las imágenes tienen el mismo mismo ancho y alto (en píxeles) que sus mapas de profundidad.
- Las imágenes a mezclar tienen el mismo ancho y alto (en píxeles).