Recuperatorio Primer Parcial

Segundo Cuatrimestre 2022

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/EiSyD5Zp27we2BgQ9`.
- Luego de la entrega habrá una instancia coloquial de defensa del trabajo

Régimen de Aprobación

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

Ej. 1 - (60 puntos)

Datos:

Un sistema de manejo de proyectos mantiene todos los proyectos registrados en una gran lista enlazada. Cada proyecto tiene un array de tareas a ser completadas y cada tarea posee una dificultad numérica (la dificultad de realizarla). Para poder visualizar rápidamente la dificultad de los proyectos se mantiene una copia precalculada de la suma de las dificultades de las tareas del proyecto. Esta copia debe ser actualizada cada vez que se modifica el proyecto.

Para mejorar el rendimiento y bajar los costos de alojamiento del sistema se solicita implementar las operaciones más usadas en ensamblador. La estructura utilizada en C es:

```
typedef struct lista_s {
        // Siguiente elemento de la lista o NULL si es el final
        struct lista_s*  next;
        // Suma de los elementos en array
        uint32_t  sum;
        // Cantidad de elementos en array
        uint64_t  size;
        // El array en cuestión que posee este proyecto de la lista
        uint32_t* array;
} lista_t;
```

Notar que:

- La lista vacía es `0` (`NULL`)
- Si `size` es 0 entonces no deberíamos leer nada del campo `array`

- Al modificar `array` no necesitamos recalcular la `sum` desde cero, alcanza con sumarle o restarle la cantidad de puntos afectados

Finalmente, como restricción para la entrega, deberán incluir la solución completa en el archivo `lista_asm.asm`, es decir su solución debe estar totalmente contenida en este archivo.

Se pide:

a. **(5 puntos)** Realizar un diagrama de la estructura en cuestión. ¿Cuál es el offset de cada campo? ¿Cuánto padding posee la estructura? ¿Cuál es su requisito de alineación?

Tabla 1

`lista_t`				
campo	tipo	tamaño	offset	alineación
next	`lista_t*`	__ bytes	__ bytes	__ bytes
sum	`uint32_t`	__ bytes	__ bytes	__ bytes
size	`uint64_t`	__ bytes	__ bytes	__ bytes
array	`uint64_t`	__ bytes	__ bytes	__ bytes
TOTAL		__ bytes		__ bytes

La estructura '`lista_t`' tiene __ bytes de padding.

b. **(20 puntos)** Implementar `uint32_t proyecto_mas_dificil(lista_t*)` que dada una lista de proyectos calcula la cantidad de puntos de el más difícil de ellos.

- La solución debe recorrer la lista pasada como parámetro y encontrar el valor más grande de los campos `sum`
- La solución no debe modificar la lista pasada como parámetro
- El proyecto más difícil de la lista vacía (`NULL`) tiene dificultad 0

Ejemplos:

- `proyecto_mas_dificil((1, 2, 3) -> (98, 99) -> (9) -> NULL) = 197`
- `proyecto_mas_dificil(() -> () -> () -> NULL) = 0`
- `proyecto_mas_dificil((3) -> (2, 1) -> (1, 1, 1) -> NULL) = 3`
- `proyecto_mas_dificil(() -> (1, 1, 1) -> (100) -> NULL) = 100`
- `proyecto_mas_dificil(NULL) = 0`

c. **(20 puntos)** Implementar `void marcar_tarea_completada(lista_t*, size_t)` que dada una lista y un índice '`i`' marca a la '`i`'ésima tarea como completada.

- La 0-ésima tarea es la primer tarea del primer proyecto no-vacío
- La lista pasada como parámetro puede tener proyectos sin tareas

Aclaraciones:

- El índice trata a la lista de arrays como si fuera un único array

  - Ej: La tarea con índice 3 de la lista `(1, 2, 3) ->() ->() ->(9) ->NULL` es aquella con dificultad 9.

- El valor de `sum` deberá reflejar los cambios

  - Ej: Luego de marcar la tarea con índice 2 de `(0, 0, 99) ->NULL` como completada la suma del primer nodo de la lista pasa de valer 99 a valer 0.

- Se puede asumir que el índice es válido

  - Ej: La tercera cuyo índice es 3 de la lista `(1, 2, 3) ->(4, 5, 6) ->NULL` es aquella con dificultad 4.

Ejemplos:

- `marcar_tarea_completada((1, 2, 3) -> NULL, 0)`
  La lista queda `(0, 2, 3) ->NULL`
- `marcar_tarea_completada(() -> (1) -> NULL, 0)`
  La lista queda `() ->(0) ->NULL`
- `marcar_tarea_completada((1, 2, 3) -> (4, 5) -> NULL, 4)`
  La lista queda `(1, 2, 3) ->(4, 0) ->NULL`
- `marcar_tarea_completada((1) -> (2) -> (3) -> NULL, 2)`
  La lista queda `(1) ->(2) ->(0) ->NULL`
- `marcar_tarea_completada((1) -> (2) -> (3) -> NULL, 1)`
  La lista queda `(1) ->(0) ->(3) ->NULL`

d. **(15 puntos)** Implementar `uint64_t* tareas_completadas_por_proyecto(lista_t*)` que dada una lista devuelve un array con la cantidad de tareas completadas en cada proyecto.

- Si se provee a la lista vacía como parámetro (`NULL`) la respuesta puede ser `NULL` o el resultado de `malloc(0)`
- Los proyectos sin tareas tienen cero tareas completadas
- Los proyectos sin tareas deben aparecer en el array resultante
- Se provee una implementación esqueleto en C si se desea seguir el esquema implementativo recomendado

Ejemplos:

- `tareas_completadas_por_proyecto(NULL) = [] ó NULL`
- `tareas_completadas_por_proyecto(() -> NULL) = [0]`
- `tareas_completadas_por_proyecto(() -> () -> NULL) = [0, 0]`
- `tareas_completadas_por_proyecto((1, 2, 3) -> NULL) = [0]`
- `tareas_completadas_por_proyecto((0, 2, 0) -> NULL) = [2]`
- `tareas_completadas_por_proyecto((1, 2) -> (3, 4) -> NULL) = [0, 0]`
- `tareas_completadas_por_proyecto((1, 2) -> (0, 0) -> NULL) = [0, 2]`

- `tareas_completadas_por_proyecto((1, 0) -> (0, 4) -> NULL) = [1, 1]`

Recomendaciones:

- Implementar la función auxiliar `uint64_t lista_len(lista_t*)` para calcular el tamaño de la lista.
- Implementar la función auxiliar `uint64_t tareas_completadas(uint32_t* array, size_t tamaño)` para calcular la cantidad de tareas completadas en uno de los arrays.

Compilación y Testeo

El `Makefile` del presente genera cuatro binarios `main_c`, `main_asm`,`tests_c` y `tests_asm`. Los primeros dos son para pruebas personales mientras que los otros dos poseen los tests de la cátedra. El sufijo `_c` y `_asm` refiere a cuál de las implementaciones está compilada en cada uno de ellos.

Los siguientes comandos están disponibles para su conveniencia:

- `make all`: Crea los binarios `main_c`, `main_asm`, `tests_c` y `tests_asm`
- `make clean`: Borra todos los archivos generados
- `make run_tests`: Compila y corre los tests usando la implementación en ensamblador.
- `make run_c_tests`: Compila y corre los tests usando la implementación en `C`.

- Los tests se ejecutan utilizando valgrind. Si no desea hacer uso de la herramienta puede ejecutar `./tests_c` manualmente
- Si valgrind no detecta errores de memoria se imprime "No se detectaron errores de memoria" en la terminal

Recomendaciones generales:

- Programar las funciones en C antes de hacerlo en ensamblador
- Asegurarse de que el código ensamblador escrito exprese la intención esperada (es decir, sea fácil de leer)
- Comentar claramente el código, de ser posible
- En caso de encontrar errores, revisar el ABI manualmente y el uso de la memoria con valgrind
- En caso de no comprender un mensaje de error de valgrind o el compilador preguntar a un docente

Ej. 2 - (40 puntos)

En este ejercicio van a programar un filtro que se aplica sobre una señal de entrada que está formada por *enteros con signo de 16 bits*. Siendo i el índice de la señal de entrada y múltiplo de 8 se pide que el filtro cumpla lo siguiente:

Señal de entrada (e):

...	$e[i]$	$e[i+1]$	$e[i+2]$	$e[i+3]$	$e[i+4]$	$e[i+5]$	$e[i+6]$	$e[i+7]$	...

Señal de salida (s):

...	$(e[i]+e[i+7])*(e[i]-e[i+4])$	$(e[i+1]+e[i+6])*(e[i+1]-e[i+5])$	$(e[i+2]+e[i+5])*(e[i+2]-e[i+6])$	$(e[i+3]+e[i+4])*(e[i+3]-e[i+7])$	$(e[i]-e[i+7])*(e[i]+e[i+4])$	$(e[i+1]-e[i+6])*(e[i+1]+e[i+5])$	$(e[i+2]-e[i+5])*(e[i+2]+e[i+6])$	$(e[i+3]-e[i+4])*(e[i+3]+e[i+7])$	...

A tener en cuenta para la resolución del ejercicio:

- El tamaño de la señal de entrada es múltiplo de 8.
- En el archivo `ej2.c` hay una implementación en *c* pueden utilizarla para guiarse con las operaciones entre los enteros.
- El archivo que deben modificar es `ej2.asm` utilizando el paradigma SIMD.
- Recuerden compilar utilizando el comando `make all`, este genera los binarios `tester` y `main`. Si lo desean, pueden correr directamente `./main` para hacer sus pruebas rápidas sin correr los tests de memoria, lo mismo para los tests de la cátedra con `./tester`. Luego, con el comando `sh runMain.sh` pueden correr sus pruebas con test de memoria y con `sh runTester.sh` los tests de la cátedra también con test de memoria.
- Este ejercicio debe realizarse utilizando el paradigma de programación SIMD, en caso contrario será considerado incorrecto.