27/04/2019

Organización de Computador (CO3815)
Prof. Kity Álvarez y Prof. Yudith Cardinale
15-10011 - Valentina Aguana Villegas
15-10420 - Luís Marcos Díaz Naranjo

Informe Proyecto I
CI3815

Introducción

El lenguaje ensamblador, o assembly (abreviado asm), es un lenguaje de programación de bajo nivel. Consiste en un conjunto de mnemónicos que representan instrucciones básicas para los computadores, microprocesadores, microcontroladores y otros circuitos integrados programables. Implementa una representación simbólica de los códigos de máquina binarios y otras constantes necesarias para programar una arquitectura de procesador y constituye la representación más directa del código máquina específico para cada arquitectura legible por un programador. Cada arquitectura de procesador tiene su propio lenguaje ensamblador que usualmente es definida por el fabricante de hardware, y está basada en los mnemónicos que simbolizan los pasos de procesamiento (las instrucciones), los registros del procesador, las posiciones de memoria y otras características del lenguaje. Un lenguaje ensamblador es por lo tanto específico de cierta arquitectura de computador física (o virtual).

Durante todo el curso, se ha trabajado con la versión MIPS Assembly del lenguaje ensamblador. MIPS (siglas de *Microprocessor without Interlocked Pipeline Stages*) es como se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies. Por su parte, en arquitectura computacional, RISC (del inglés Reduced Instruction Set Computer, en español Computador con Conjunto de Instrucciones Reducidas) es un tipo de diseño de CPU generalmente utilizado en microprocesadores o microcontroladores con las siguientes características fundamentales: instrucciones de tamaño fijo y presentadas en un reducido número de formatos; solo los comandos de carga y almacenamiento acceden a la memoria de datos.

Para programar utilizando assembly se utilizan hoy en día simuladores como MARS, que es un emulador libre, gratuito y multiplataforma programado en Java que permite ejecutar código de la arquitectura de procesadores MIPS, ver su funcionamiento interno en los registros del sistema, y guardar el código en ficheros externos. MARS será el simulador seleccionado para la realización de todo el proyecto.

Si bien los lenguajes de bajo nivel pueden parecer restringidos en comparación con los lenguajes de programación modernos, las mismas estructuras utilizadas en estos últimos pueden reproducirse en Assembly. En este caso, se implementó un manejador de memoria dinámica, que además permite la creación por parte del usuario listas enlazadas.

Implementación

El proyecto se encuentra dividido en dos partes: el TAD Manejador y el TAD Lista. El primero consiste en un manejador de memoria dinámica mientras que el segundo se trata de la implementación de una lista enlazada, aprovechando las funciones previamente creadas para la primera parte del proyecto.

TAD Manejador (memory-manager.asm):
El objetivo es recrear el manejador de memoria ya implementado en MARS (Syscall 9) para así administrar un espacio fijo de memoria donde se ofrezca a las aplicaciones un servicio de reserva. A su vez, se debe crear una nueva función para la liberación de espacio para así conseguir manejar la memoria de forma dinámica.

Para implementar cada una de las funciones necesarias, se estableció una estructura abstracta para la representación virtual del heap, basándose en la forma en que se ocupan las direcciones de memoria solicitadas por el usuario. Esta estructura está dividida en tres partes: head, espacio reservado y tail.

Ej: n=3, m=2

| ***Head:*** n+1| 1 | 1 | ***Tail:*** -(n+1) | 0 | 0 | ***Head:*** m + 1 | 1 | 1 | ***Tail:*** -(m+1) |

Una head o una tail tienen valor n+1, donde n es la cantidad de espacio alojado para ese malloc. En caso de ser una head dicho valor es positivo, en caso de ser un tail es negativo.

Funciones del TAD Manejador:

`Init:` reserva una cantidad de espacio dada por el usuario que servirá como base para el espacio total disponible a utilizar por otras funciones. Esta reserva de memoria no es dinámica. Se estableció un heap virtual de tamaño 500, por lo que si se realiza un `Init` de mayor tamaño, la función arrojará error.

- Entrada: un entero menor o igual a 500.

- Salida: la dirección donde comienza la memoria reservada (`InitHead`). En caso de superar el tamaño permitido por el heap, devuelve -1.

`Malloc:` permite el manejo de memoria dinámica por el usuario, por lo que reserva una cantidad de espacio deseado dentro del segmento previamente guardado por el `Init`. Dicho espacio reservado puede ser liberado en el futuro gracias a la función `free`.

Malloc empieza en el inicio del segmento de memoria inicializada por `Init` y se pregunta si el contenido de la palabra (4 bytes) en la que se encuentra es vacío (cero), y en caso afirmativo suma uno al contador de espacio disponible. De lo contrario, reinicia el contador y salta a la siguiente posible dirección disponible. Para saltar, usa el valor contenido en esa palabra no vacía, pues era un `head` con la información de cuántas palabras alojadas tiene delante. En caso de que el contador nunca alcance la cantidad solicitada, arroja un error por falta de suficiente espacio contiguo.

- Entrada: cantidad de bytes que se esperan alojar.
- Salida: Posición de memoria donde empieza el segmento con la cantidad de bytes pedidos. Si la cantidad solicitada no es múltiplo de 4, se completa con una palabra adicional (padding). En caso de no encontrar espacio disponible, devuelve un -2.

`Free:` permite liberar memoria anteriormente reservada por `malloc`. La función recibe una dirección de memoria con la cabeza del segmento de memoria que se desea liberar. Al igual que el malloc, revisa que la palabra en la que se encuentra sea un `head`, ya que de ser así, procede a vaciar la cantidad de palabras consecutivas indicadas por dicho head. En caso de que la dirección no corresponda a una `head`, arroja un error.

- Entrada: dirección de memoria correspondiente al inicio del segmento de memoria que se desea liberar.
- Salida: si la función culmina con éxito, devuelve un 1, de lo contrario arroja un -3.

`Perror:` maneja los posibles errores arrojados por las diferentes funciones, incluyendo posibles errores en la función `delete` correspondiente al TAD Lista.

- Entrada: un entero negativo correspondiente al código de error correspondiente
- Salida: void

Tabla de errores:

Código de error	Función	Print
-1	`Init`	“Error. La memoria solicitada supera el almacenamiento del heap”
-2	`Malloc`	"Error. No hay espacio suficiente en memoria para alojar la cantidad solicitada"
-3	`Free`	"Error. La dirección ingresada no es un head"
-4	`Delete`	“Error. El elemento no se encuentra en la lista"
Otro	-	“Unknown error”

TAD Lista (linked-list.asm):

El objetivo es crear una lista enlazada simple, compuesta por dos estructuras principales: la cabeza de la lista y los nodos. La head de la lista está conformada por un apuntador al primer elemento de la lista, la cantidad total de nodos existentes y un apuntador al último nodo. Por otra parte, los nodos están conformados por un apuntador al siguiente nodo y un apuntador a la dirección de memoria donde se encuentra su contenido. Es importante resaltar que tanto la head como los nodos de la lista tienen un tamaño fijo, sin embargo el contenido de cada nodo puede tener virtualmente cualquier tamaño debido a que únicamente se aloja su dirección.

Funciones del TAD Lista:

`Create:` instancia una lista y devuelve la posición de memoria donde está la cabeza de la misma. La lista inicializada posee 3 atributos: “inicio”, la dirección donde se encuentra el primer nodo de la lista; “fin”, la dirección del último nodo de la lista; y “size”, la cantidad de nodos actuales de la lista.

Para inicializar la lista, se deben alojar 12 bytes de espacio: 4 para el valor del atributo "inicio", 4 para el valor del atributo "fin" y 4 para el número de elementos. "Inicio" y "fin" empiezan valiendo 0x0 originalmente, y el número de elementos comienza en 0.

|4 bytes: inicio | 4 bytes: fin | 4 bytes: size |

- Entrada: void
- Salida: dirección donde se encuentra la cabeza de la lista.

`Insert:` dada una lista, inserta un elemento mediante la creación de un nodo que posee la dirección asociada a dicho elemento. Cada nuevo nodo posee dos atributos: “dir”, dirección en la memoria donde se encuentra el elemento correspondiente con ese nodo; “next”, dirección del siguiente nodo en la lista. Si el valor de siguiente es 0x0, entonces no tiene siguiente y por lo tanto este nodo es la tail (final) de la lista.

Se instancia el nodo en cuestión utilizando un malloc(8) y luego asignando el contenido de sus bytes como se indica en la estructura. Este nodo se asigna como siguiente del nodo "final". Si es el primero en ser añadido, entonces también se define como head y tail al mismo tiempo, ya que el siguiente elemento de la lista sería vacío.

|4 bytes: dirección elemento | 4 bytes: next|

- Entrada: “lista_ptr”, apuntador al head de la lista; “dir”, dirección del elemento a añadir en la lista.
- Salida: 1

`Delete:` elimina de la lista el nodo indicado. Se implementa la eliminación usual de las listas enlazadas, por lo que se consideran varios casos bordes: en el caso donde se pide una posición que excede el tamaño de la lista, el programa genera un error. En el caso donde se elimina el único elemento de la lista, ésta se vacía. En el caso donde se elimina el primero o el último de la lista, estos atributos son actualizados en la head de la lista.

- Entrada: “lista_ptr”, apuntador al head de la lista; “pos”: la posición del elemento en la lista (primero, segundo, tercero...).
- Salida: si el delete fue exitoso, devuelve la dirección del elemento que fue eliminado. Por el contrario, en caso de error regresa un -4.

`Print:` imprime todos los elementos de la lista enlazada, haciendo uso de la función auxiliar `fun_print`. Se guarda una variable con la dirección del siguiente elemento a imprimir, y mientras esta dirección sea distinta de 0x0, se llama a la función de impresión "function" sobre esa dirección, luego se imprime un espacio.

- Entrada: “lista_ptr”, apuntador al head de la lista; function(): función utilizada para imprimir un nodo. Necesitamos diferentes funciones dependiendo del tipo de la lista porque el tipo de dato que almacena esta lista cambia el método de representación.
- Salida: void

`Fun_print:` Utilidad auxiliar para imprimir enteros en una lista enlazada. Se carga la dirección almacenada en el nodo, y luego el contenido en esa dirección en $a0 y se imprime.

- Entrada: “nodeDir”: dirección del nodo que queremos imprimir
- Salida: void