Universidad Simón Bolívar
Departamento de Computación y T.I
CI-3815, Organización del Computador
Enero-Marzo 2019

Proyecto # 1

Objetivo

Adquirir destrezas básicas de programación en lenguaje MIPS y representación de estructuras de datos.

Objetivos Específicos:

- Adquirir destrezas en el lenguaje de bajo nivel usando el simulador MARS
- Aprender a programar subrutinas o funciones en el lenguaje de bajo nivel
- Aprender a usar convenciones para el uso de subrutinas/funciones
- Adquirir destrezas en el uso de estructuras de datos en el lenguaje de bajo nivel

Enunciado

Los recursos del computador son administrados por un grupo de programas con los cuales los usuarios se comunican para indicarle qué operaciones se requiere realizar. MARS permite comunicarle a estos programas estas operaciones a través de las llamadas al sistema (syscall). Una de estas llamadas permite al usuario solicitar la asignación de un espacio de memoria en forma dinámica (Syscall 9) para la realización de alguna actividad. En la actualidad no existe, en MARS, una llamada al sistema que permita a un usuario devolver al sistema de operación los espacios de memoria que hayan sido solicitados y que ya no sean necesario mantener por parte de la aplicación. En otras palabras, los programadores pueden pedir asignación de memoria en forma dinámica pero no tienen un mecanismo para devolverla.

Este proyecto consiste en implementar un manejador de memoria principal que le permita a los usuarios realizar dos operaciones:

1. reservar espacios de memoria,
2. redimensionar espacios de memoria,
3. liberar espacios de memoria previamente reservados

Adicionalmente, para facilitar la verificación del correcto funcionamiento de este manejador, se debe implementar el tipo abstracto de dato Listas.

Actividades

Actividad 1: TAD_Manejador

El TAD_Manejador tendrá como función el administrar un espacio fijo de memoria para ofrecer a las aplicaciones el servicio de reserva y liberación de espacios en forma dinámica. Este TAD debe ser implementado en función de su representación y sus operaciones.

Representación

La representación de datos queda por parte de UD. debe definir las estructuras que considere más

pertinente. Debe, además, comentar brevemente sobre sus ventajas y desventajas en función de las operaciones definidas.

Operaciones

1. **init(IN size:entero; OUT code: entero)**

Pre:	true
Post:	Las estructuras que componen la representación del TAD se encuentran inicializadas. El manejador de memoria será capaz de administrar en total hasta *size* bytes.
Parámetros:	
size:	Número de bytes asignados al manejador para ser administrados. **Esta función debe solicitar un bloque de memoria de *size* bytes y de este bloque es que él posteriormente podrá seleccionar segmentos para asignarlos a los programas que lo soliciten a través de la función malloc.**
Retorno:	
code	0 si la operación se realizó correctamente.
	Valor negativo que representa el código del error ocurrido.

2. **malloc(IN size:entero; OUT address: entero)**

Pre:	Las estructuras que componen la representación del TAD se encuentran inicializadas.
Post:	Las estructuras que componen la representación del TAD se actualizan reflejando la asignación de *size* bytes de memoria, en caso de que el Manejador disponga de size bytes consecutivos libres.
Parámetros:	
size:	Número de bytes solicitados
Retorno:	
address	dirección de memoria donde comienza *los size* bytes consecutivos que fueron reservados para el invocador de la función
	Valor negativo que representa el código del error ocurrido.

3. **reallococ(IN direc; dir, IN size:entero; OUT address: entero)**

Pre:	Las estructuras que componen la representación del TAD se encuentran inicializadas.
Post:	Las estructuras que componen la representación del TAD se actualizan reflejando la re-asignación de *size* bytes de memoria, en caso de que el Manejador disponga de size bytes consecutivos libres. Note: • **caso 1**: el valor *size* sea menor o igual al número de bytes asociado con *direc* entonces el manejador debe ser acualizado con el nuevo número de bytes • **caso 2**: el valor *size* sea mayor al número de bytes asignados previamente a *direc*, el manejador debe ubicar un nuevo

	segmento de bytes libres y copiar el número de bytes previamente asignados a *direc* al nuevo espacio asignado.
Parámetros:	
direc	Dirección de memoria previamente asignada por el manejador. Es decir, el manejador debe tener registro del tamaño asociado a esta dirección.
size:	Nuevo número de bytes solicitados, es decir, el tamaño que se solicita sea asignado. Este valor puede ser mayor o menor al previamente asignado.
Retorno:	
address	• **Caso 1:** address será igual a direc
	• **Caso 2:** nueva dirección de memoria donde comienza *los size* bytes consecutivos que fueron reservados.
	Valor negativo que representa el código del error ocurrido.

4. **free(IN address:entero; OUT code: entero)**

Pre:	Las estructuras que componen la representación del TAD se encuentran inicializadas. address corresponde a un segmento de memoria previamente reservado y aún no liberado.
Post:	El segmento de memoria referido por address vuelve a estar disponible para una próxima reserva.
Parámetros:	
address	Dirección de comienzo en memoria del segmento de datos que se requiere liberar.
Retorno:	
code	0 si la operación se realizó correctamente.
	Valor negativo que representa el código del error ocurrido.

5. **perror(IN code:entero; OUT void)**

Pre:	true
Post:	true
Parámetros:	
code	valor negativo que identifica un error ocurrido. Debe imprimir el mensaje asociado al valor *code. Si* el valor es positivo esta función no debe imprimir nada.
Retorno:	void

Notas

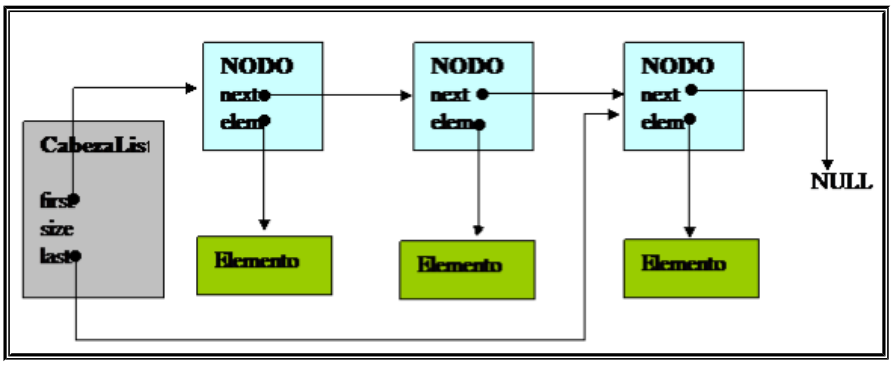
- Ud debe definir una tabla de códigos de error y sus mensajes correspondientes e incluirla en el informe. En la documentación del código debe incluir los valores de error que retorna cada función.
- Sólo la función perror puede imprimir por cónsola mensajes, las otras operaciones de limitaran a devolver un código de error apropiado.

Actividad 2: TAD Lista

Implementar el TAD Lista en función de su representación y sus operaciones.

Representación

Debe ser implementada como una Lista enlazada simple como se muestra en el diagrama siguiente. Queda a su elección la variación específica a usar. Debe ser implementada usando memoria dinámica a través de los servicios ofrecidos por el TAD_Manejador.



Donde:

- **CabezaLista:** es definido de la siguiente forma:

```
CabezaLista: Registro
                        Nodo_ptr first;
                        Nodo_ptr last;
                        entero   size;
             Fin.
```

  - first: dirección donde se encuentra almacenado el primer Nodo
  - last: dirección donde se encuentra almacenado el último Nodo
  - Size: Número de elementos en la lista.
- **Nodo:** es un Elemento que contiene:

```
Nodo: Registro
                  Nodo_ptr     next;
                  Elemento_ptr elem;
      Fin.
```

  - next: dirección del siguiente nodo, en caso de ser el último, el valor que contendrá debe ser 0x0 (NULL).
  - elem: dirección donde se encuentra almacenado el elemento.

- **Elemento:** es una estructura genérica externa que representa el dato que se desea almacenar en la lista. Elemento puede verse como una estructura abstracta (como se usa en lenguaje Java) que otras estructuras deberán extender (extends en Java). Toda estructura que *extienda* de un Elemento debe definir una función:

  - **print**: función que define cómo la estructura extendida puede ser impresa.

  A continuación algunos **ejemplos** de estructuras que podrían extender de la estructura Elemento.

  - **Estudiante extiende Elemento**

```
Estudiante: Registro
                  String Nombre;
```

```
                String CI;
                entero Edad;
            Fin.
```

- **Materia extiende Elemento**

```
Materia: Registro
                String Nombre;
                String Codigo;
                entero numero_creditos;
        Fin.
```

Operaciones

1. **create(OUT: address: entero)**

Pre:	True
Post:	La lista está inicializada. address contiene la dirección que referencia a la cabeza de lista.
Parámetros:	void
Retorno:	
address	dirección donde se encuentra la cabeza de lista.
	Valor negativo que representa el código del error ocurrido.

2. **insert(IN lista_ptr: entero; IN elem_ptr: entero; OUT code: entero)**

Pre:	Las estructuras que componen la representación del TAD se encuentran inicializadas.
Post:	el elemento referido por elem_ptr forma parte de la lista y será colocado como último elemento. El valor de size se incrementa en 1.
Parámetros:	
lista_ptr	dirección de la cabeza de la lista.
elem_ptr	dirección del elemento a ser insertado.
Retorno:	
code	0 si la operación se realizó correctamente.
	Valor negativo que representa el código del error ocurrido.

3. **delete(IN lista_ptr: entero; IN pos: entero; OUT address: entero)**

Pre:	Las estructuras que componen la representación del TAD se encuentran inicializadas. pos <= size. Los elementos de la lista están en las pocisiones [1...size]. No hay elemento en la posición 0.
Post:	El elemento en la posición pos ya no forma parte de los elementos de la lista. Es liberado el segmento correspondiente al Nodo correspondiente.
Parámetros:	
lista_ptr	dirección de la cabeza de la lista.

pos	posición del elemento que se desea liberar.
Retorno:	
address	Dirección de memoria del Elemento_ptr corrspondiente.
	Valor negativo que representa el código del error ocurrido.

4. **print(IN lista_ptr: entero; IN fun_print: entero; OUT void)**

Pre:	Las estructuras que componen la representación del TAd se encuentran inicializadas.
Post:	true
Parámetros:	
lista_ptr	dirección de la cabeza de la lista.
fun_print	Es la dirección de la función para imprimir los elementos. La lista se imprime por consola. Cada elemento es impreso usando esta función. **fun_print: (IN a: elemento; OUT void)** donde: • a es una dirección de memoria del elemento a imprimir.
Retorno:	void

Ejemplo de un programa en alto nivel que hace uso de los TADs descritos en las actividades 1 y 2.

```
main (){
      int error = 0;
      int lista_ptr;
      int elemento_ptr;

      error = init(4096);

      if (error < 0) {
          perror(error);
          exit();
       }

      lista_prt = create();

      if (lista_prt < 0) {
          perror(error);
          exit();
       }

       for (int i = 0; i < 10; i++) {
          elemento_ptr = elem_crea_reamdom();
          error = insert(lista_prt, elemento_prt);

          if (error < 0) {
              perror(error);
             .....
          }
       }
```

```
      print(lista_prt, elem_print);
}

elemento_prt elem_crea_random() {
      ...........
}

void elem_print(elemento_ptr e) {
      ...........
}
```

Nota: por simplicidad de **ESTE** ejemplo, en no se está verificando si hubo error en algunas funciones. Los casos de prueba en la corrida de este proyecto usarán las funciones tal y como son definidas en las actividades 1 y 2.

Recomendaciones

1. Trabaje en forma ordenada e incremental
2. Pruebe que cada una de sus funciones funciona correctamente
3. Estructure bien su código.
4. Tenga presente que es mejor tener más funciones pequeñas que menos funciones largas.

Actividad 3: Entrega

El proyecto deberá ser enviado como un archivo .tar.gz a ycardinale@usb.ve, con el asunto siguiendo el siguiente formato: [Organizacion] Proyecto 1 – Carnet1-Carnet2

Día: Jueves 7 de marzo 11:59pm (semana 6).