Arquitecturas Intel 64 e IA-32

Arquitectura y Organización del Computador

Actividad Individual

La presente es una guía de ejercicios preparatorios para el parcial individual. Hay ejercicios de diferente grado de dificultad que buscan practicar la programación en Assembler y C, y repasar los conceptos teóricos relevantes para la parte práctica de la materia.

Indicamos con estrellas el nivel de dificultad relativo. No es necesario que completen toda la guía pero si que traten de elegir los ejercicios en función de lo que más comprenden o no. Es decir, si se sienten seguros pueden hacer sólo los difíciles, pero si hay temas que no les cierran pueden elegir los más básicos y luego pasar a los intermedios o difíciles.

1. Guía de evaluación

Esta es una breve enumeración de algunos temas que vimos en la práctica y nos interesa que hayan comprendido. No son los únicos temas que se van a evaluar pero les pedimos que en su preparación para el examen se aseguren manejar los siguientes.

- Armado de stackframe
- Alineación de la pila en 32 y 64 bits
- Convención C en 32 y 64 bits:
  - Registros a preservar
  - Pasaje de parámetros a función en 32 y 64 bits
- Llamadas a función con CALL y retorno con RET impacto en la pila
- Saber dibujar todo lo que se guarda en la pila en llamadas anidadas y no anidadas a función en C y Assembler
- Conversión de enteros a float y double para operar matemáticamente
- Manejo de datos:
  - Lectura/escritura/almacenamiento de distintos tamaños de datos (8, 16. 32 y 64 bits).
  - Manejo de caracteres, strings, enteros, floats y doubles.
  - Tamaño de los registros y lecturas a memoria con tamaño adecuado al dato.
- Punteros en C

2. Ejercicios

2.1. C y Assembler

En la siguiente, sección deben escribir código en C y Assembler desde cero. Recuerden agregar en C el prototipo de las funciones que escriban en Assembler para poder llamarlas.

Observen que las funciones en C tienen la palabra `extern` y en Assembler, `global`. Esto permite exponer funciones de C a Assembler y viceversa.

Ejercicio 1 ⋆⋆
Dada una cadena de texto s definiremos su *codificación césar simplificada* respecto de un entero x de la siguiente manera:

La función recibe únicamente caracteres en mayúsculas (es una precondición). Para explicarlo supongamos la existencia de dos funciones: `ord` y `chr`, ambas definidas para las mayúsculas únicamente.

- `ord(c)` es el número de letra de c. (A = 0, Z = 25)
- `chr(n)` es la letra correspondiente al número n.
  La función "vuelve a empezar desde el princpio" cuando se queda sin letras, o sea `chr(26k + n) == chr(n)`.

Con estas dos funciones el pseudocódigo de `cesar(s, x)` es:

```
resultado := una cadena con tamaño longitud(s)
por cada i de 0 a longitud(s):
  la i-ésima letra del resultado es chr(ord(c) + x)
devolver resultado
```

Ejemplo:

- `cesar("CASA", 3) = "FDVD"`
- `cesar("CALABAZA", 7) = "JHSHIHGH"`

Más información: https://es.wikipedia.org/wiki/Cifrado_César

Puede asumir que $0 \leq x < 26$. Opcionalmente resuelva el problema general.

Recuerden que todos los bytes que representan las letras estan dadas por la codificación ASCII. Por ejemplo, el 65 corresponde al caracter `A` mayúscula. Fijense cual sería el valor binario que tomaría cada letra mayúscula en la codificación ASCII.

a) Escribir una función en C que dada una cadena de caracteres y un entero, devuelva su codificación césar simplificada.

b) Realice otra implementación en assembler respetando el ABI visto en la cátedra. Cree un pequeño programa en C que utilice esta implementación.

Ejercicio 2 ⋆
Escribir un programa en C que dados dos strings determine la longitud de su prefijo común más largo.

Ejemplos:

- `prefijo_de("Astronomia", "Astrologia") = 5` // ("Astro")
- `prefijo_de("Pinchado", "Pincel") = 4` // ("Pinc")
- `prefijo_de("Boca", "River") = 0` // ("")
- `prefijo_de("ABCD", "ABCD") = 4` // ("ABCD")

Opcional Escribir un programa que dados dos strings devuelva "el segundo sin el prefijo común con el primero".

Para construir un nuevo string deberán hacer uso de memoria dinámica (`malloc`/`free`).

Ejemplos:

- `quitar_prefijo("Astro", "Astrologia") = "logia"`
- `quitar_prefijo("Pinchado", "Pincel") = "el"`
- `quitar_prefijo("Boca", "River") = "River"`
- `quitar_prefijo("ABCD", "ABCD") = ""`

Ejercicio 3 ⋆⋆
Teniendo en cuenta las siguientes estructuras en C:

```
#define NAME_LEN    21

typedef struct cliente_str {
    char nombre[NAME_LEN];
    char apellido[NAME_LEN];
    uint64_t compra;
    uint32_t dni;
```

```
} cliente_t;

typedef struct  __attribute__((__packed__)) packed_cliente_str {
    char nombre[NAME_LEN];
    char apellido[NAME_LEN];
    uint64_t compra;
    uint32_t dni;
} __attribute__((packed)) packed_cliente_t;
```

a) Dibuje un diagrama de organización en memoria de las estructuras `cliente_t` y `packed_cliente_t` marcando el espacio usado para alineación y padding.

b) Escriba un programa en Assembler que dado un arreglo `cliente_t[]` con su longitud devuelva un puntero a alguno uno de sus elementos al azar.

Tip Para la generación de números aleatorios puede utilizar la función `rand` provista por la biblioteca estándar de C.

Ejercicio 4 ⋆⋆
Una lista enlazada es una estructura de datos que nos permite almacenar datos como una secuencia de nodos. Cada nodo consiste de su valor y un puntero al que nodo que viene después. Lo único que se necesita para tener acceso a toda la lista es un puntero al nodo que la encabeza (llamado `head`).

Considere la siguiente estructura y sus operaciones:

```
typedef struct node_str {
    struct node_str* siguiente;
    int32_t valor;
} node_t;

node_t* agregarAdelante(int32_t valor, node_t* siguiente);
node_t* agregarAdelante_asm(int32_t valor, node_t* siguiente);
int32_t valorEn(uint32_t indice, node_t* cabeza);
void destruirLista(node_t* cabeza);
```

a) Implemente `agregarAdelante(int32_t, node_t*)` de forma que la expresión `head = agregarAdelante(123, head);` nos permita agregar el valor 123 al principio de la lista llamada `head`.

b) Implemente `agregarAdelante_asm(int32_t, node_t*)` en lenguaje ensamblador.

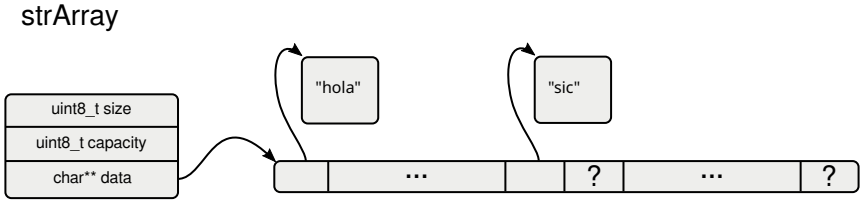
Opcional ⋆⋆⋆ Implemente `valorEn(uint32_t, node_t*)` en Assembler de forma que nos permita obtener el iésimo elemento de la lista.

Opcional ⋆⋆⋆ Implemente `destruirLista(node_t*)` en Assembler, que dada la cabeza de la lista elimina toda la lista. Corrobore su correcto funcionamiento utilizando `valgrind`.

2.2. Ejercicios de Parcial

Ejercicio 5 Implementa un array dinámico de capacidad limitada. El mismo puede contener como máximo la cantidad de strings indicada en `capacity`. Los datos serán todos del tipo *String*

La estructura `str_array_t` contiene un puntero al arreglo identificado como *data* y la cantidad de elementos ocupados como *size*.




```
typedef struct str_array {
  uint8_t size;
  uint8_t capacity;
  char** data;
} str_array_t;
```

Implementar las siguientes funciones en asm:

- `str_array_t* strArrayNew(uint8_t capacity)`
  Crea un array de strings nuevo con capacidad indicada por `capacity`.
- `uint8_t strArrayGetSize(str_array_t* a)`
  Obtiene la cantidad de elementos ocupados del arreglo.
- `char* strArrayGet(str_array_t* a, uint8_t i)`
  Obtiene el i-esimo elemento del arreglo, si `i` se encuentra fuera de rango, retorna `NULL`.
- `char* strArrayRemove(str_array_t* a, uint8_t i)`
  Quita el i-esimo elemento del arreglo, si `i` se encuentra fuera de rango, retorna `NULL`. El arreglo es reacomodado de forma que ese elemento indicado sea quitado y retornado.
- `void strArrayDelete(str_array_t* a)`
  Borra el arreglo, para esto borra todos los strings que el arreglo contenga, junto con las estructuras propias del tipo arreglo.

Ejercicio 6 Considerar una estructura de lista doblemente enlazada en donde cada nodo almacena un string de C, es decir, un arreglo de caracteres finalizado en el caracter nulo (‘\0’).

```
typedef struct node_t {
    struct node_t *next;
    struct node_t *prev;
    char *string;
} node;
```

a. Escribir en ASM una función que reciba como parámetros `n`: *doble puntero a nodo* y lo borre de la lista junto a su string. Esta función se puede usar tanto para borrar el primer elemento de la lista (`borrarNodo(&lista);`) como para borrar algún otro (`borrarNodo(&lista->next->next);`), suponiendo que se pasa un puntero al campo next del nodo anterior.

   Utilizar la función `free` para borrar tanto los nodos como las strings.

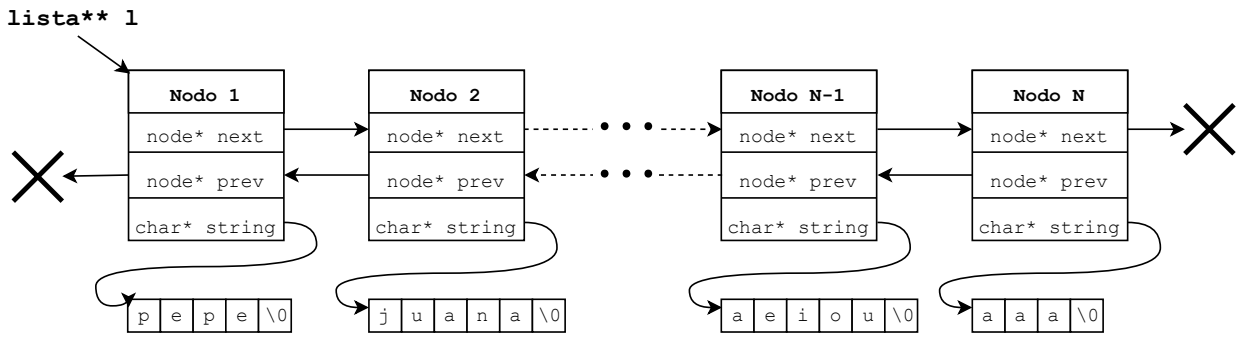
b. Escribir en ASM una función que reciba como parámetros `l`: *doble puntero a nodo* y `c`: *un caracter*, y borre todos los nodos, junto con la string, para los nodos donde que el último caracter del string sea `c`.

   Utilizar la función del item anterior para borrar los nodos.

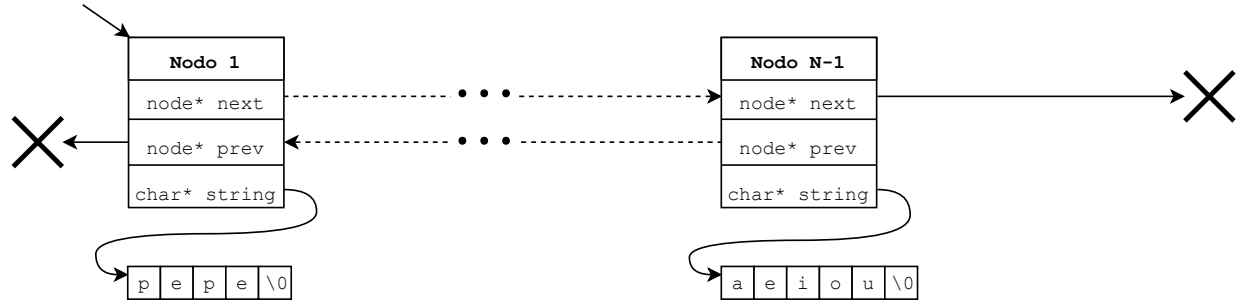
   Su aridad es: `void borrarTerminadasEn(node** l, char c)`.

   En el caso de borrar el primer elemento de la lista se debe actualizar el puntero recibido.

Resultado de aplicar borrarTerminadasEn(l, 'a')



lista l - luego de ejecutar borrarTerminadasEn(l, 'a')**



c. Escribir en ASM la función `char* superConcatenar(node* n)`, que toma un puntero a un nodo cabeza de lista y retorna una nuevo cadena que contiene la concatenación de todas las strings almacenadas en la lista.

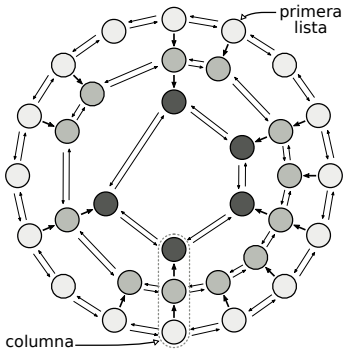
Se cuenta con la función `uint32_t strlen(char* s)` que dada una string, retorna la cantidad de caracteres válidos de la misma.

Ejercicio 7 Sea la siguiente estructura de listas doblemente enlazadas encadenadas entre si.

```
struct supernode {
  supernode* abajo;
  supernode* derecha;
  supernode* izquierda;
  int dato;
};
```

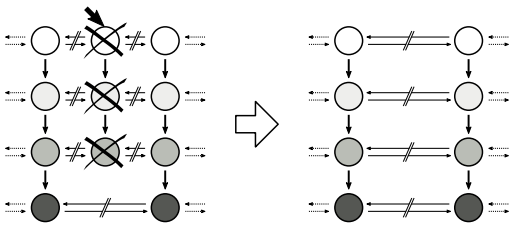
- todos los nodos pertenecen a una lista doblemente enlazada
- todos los nodos son referenciados desde algun otro nodo en otra lista (excepto en la primera)
- todas las listas respetan el orden de los nodos que las apuntan

ejemplo

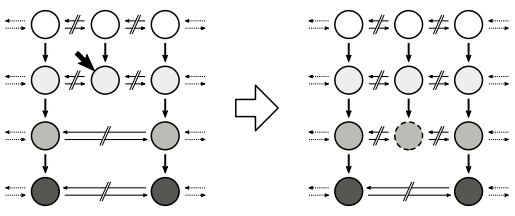


Implementar en ASM las siguientes funciones.

a. `void borrar_columna(supernode** sn)`: Dada un doble puntero a nodo dentro de la primera lista, borra una columna de nodos. Modifica el doble puntero dejando un nodo valido de la primer lista.



b. `void agregar_abajo(supernode** sn, int d)` Agrega un nuevo nodo a la lista inmediata inferior del nodo apuntado. Considerar que el nodo donde agregar puede no tener vecinos inmediatos en la lista inferior.



2.3. Ejercicios C y Assembler (Avanzados)

Se incluye el código para completar los siguientes ejercicios que permiten un mejor entedimiento del funcionamiento de la pila.

Ejercicio 8 Tamaño del stack ⋆ ⋆ ⋆ ⋆

a) Observe y determine qué hace la función de `test_medir_stack_C` definida en `medir-stack/main.c`.

b) Suponga que se ejecuta `test_medir_stack_C(3)` desde `main_en_c`. Dibuje cómo quedaría compuesta la pila al final de la tercer llamada recursiva.

c) Implemente la función `medir_stack_asm` que cuenta las apariciones del RBP en la pila hasta que encuentra el marcador 0.

Ejercicio 9 Try ⋆ ⋆ ⋆ ⋆ ⋆
Basándonos en las ideas del ejercicio anterior implementaremos un pequeño mecanismo de manejo de errores. Observemos las dos funciones a implementar:

```
/**
 * Intenta llamar a ‘funcion‘ pasándole ‘ctx‘ como parámetro.
 *
 * Si ‘función‘ produce algún error devuelve el código de error
 * correspondiente (0 indica la ausencia de error).
 */
int try(void* ctx, void (*funcion)(void* ctx));

/**
 * Falla la ejecución actual.
 *
 * En caso de que la ejecución actual esté contenida por un ‘try‘ entonces
 * ‘error_code‘ será su resultado.
 */
void fail(int error_code);
```

La idea general es que `try` deja una marca reconocible en la pila la cuál es buscada por `fail` cómo punto de recuperación ante una falla grave. Esto último es muy interesante: `fail` puede "descartar"tantos stackframes cómo considere necesarios con tal de encontrar un punto de recuperación.

El mecanismo es similar a otros como try/catch, setjmp/longjmp y panic/recover. La implementación dada cubre casos de uso básicos y no representa una solución de manejo de errores robuta en todos los casos.

El uso general de estas funciones es el siguiente:

```
  // Del lado del manejo de errores
  int error_code = try(parametro, cosa_que_puede_fallar_terriblemente);
  if (error_code != 0) {
    // Se produjo un error y nos toca intentar poner el programa en un estado
    // predecible
  }

  // Del lado dónde se generan errores
  if (validar(cosa) == ROMPER_TODO) fail(ERROR_ROMPER_TODO);
```

La utilidad del modelo de manejo de errores es que permite centralizar los puntos de recuperación por fuera del flujo normal del programa. La ausencia de mecanismos como éste normalmente resulta en funciones con la siguiente estructura:

```
int hacer_cosa0() {
  FILE* archivo = fopen("miarchivo.txt");
  if (archivo == NULL) return ERR_OPEN; // No pude leer el fichero
  char* buffer = malloc(sizeof(char[256]));
  if (buffer == NULL) return ERR_MALLOC; // No pude pedir memoria
  int codigo = leer_datos(archivo, buffer);
```

```
  if (codigo == ERR_LECTURA) return ERR_LECTURA; // No pude leer cosas
  ...
}

int leer_datos(FILE* archivo, char buffer[256]) {
  int codigo = inicializar_sistema_si_no_se_inicializo();
  if (codigo != TODO_OK) return ERR_INIT; // No pude inicializar cosas
  ...
}
```

Como puede verse, el chequeo de errores termina predominando por sobre la función del programa[1].

Entonces, para construir nuestro mecanismo revolucionario de manejo de errores tenemos que aprender a hacer dos cosas:

1. Encontrar el `try` más interno en el stack actual.
2. Retornar a un stackframe que no es el que llamó a `fail`.

El código a medio implementar se encuentra en `src/try`, los ejercicios propuestos son los siguientes:

a) Haga que `try` retorne 0 en caso de no reportarse errores.

b) Complete el epílogo de `try`.

c) Complete la búsqueda del stackframe de la función que ejecutó `try` en `fail`.

d) Observe las últimas instrucciones de `fail`:

   a) ¿Qué pasa con la pila?

   b) ¿Por qué la función no tiene `ret`? ¿Qué pasaría si tuviera un `ret` en lugar de ese `jmp`?

e) Realice una serie de diagramas de pila que comparen cómo se ejecuta bajo `try` una función que no falla y una que llama a `fail`.

f) Busque la documentación de las funciones `setjmp`/`longjmp` del lenguaje C. ¿En qué difieren de `try`/`fail`?

Ejercicio 10 Considerar un árbol binario de búsqueda con la siguiente estructura.

```
typedef struct node_t {
  node *izquierda;
  int value;
  node *derecha;
} node;
```

y un conjunto de funciones con la siguiente aridad: `typedef int func(node *a, node *b)`

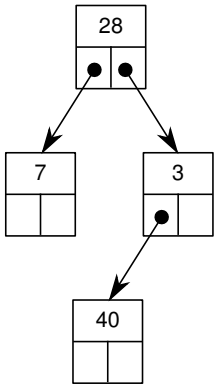
a) Escribir en ASM una función `void expandir(node* arbol, func f)` que recorra todo el árbol evaluando `f` para cada par (padre, hijo izquierdo) del árbol, agregando un nodo entre ambos con el resultado.

b) Construir en ASM una función `void comprimir(node *nodo)` para recorra todo el árbol realizando la siguiente acción: si el hijo de un nodo tiene exactamente un hijo, entonces será eliminado (conectando al nodo con su nieto). Luego de eliminar el hijo de un nodo, es posible que el nuevo hijo (que era nieto) también tenga exactamente un hijo. Ésta operación debe repetirse hasta que no se cumpla la propiedad.

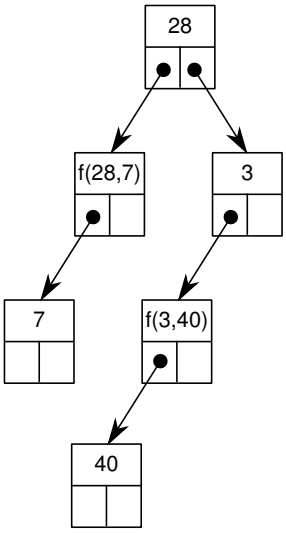
   Recomendación: utilizar una función auxiliar `void comprimir_rama(node **ppHijo)`, que comprima una rama lineal del árbol (compuesta por nodos de un hijo), hasta encontrar el primero que tenga 0 o 2 hijos.

[1] Obviamente este es un ejemplo exagerado. Hay comunidades dónde este mecanismo de manejo de errores es visto como la forma "correcta".

Árbol original



Árbol expandido



Árbol expandido, luego comprimido