Arquitectura de las Computadoras

Trabajo Práctico Nº 3

Estructura de datos en assembly del MIPS R2000

Ing. Walter Lozano
Ing. Alejandro Rodríguez Costello

Cuando nos presentaron el mapa de memoria del MIPS R2000 observamos que existen tres regiones notorias muy importantes a las que denominamos segmentos. El nombre estos son *text segment*, *data segment* y el *kernel segment*. Con la directiva **.text** inicializamos el text segment, con **.data** el espacio de direccionamiento *static data segment* y el kernel segment queda fuera del alcance de este curso. Adicionalmente aprendimos tambien a manipular el *stack* con el puntero de pila **$sp**.

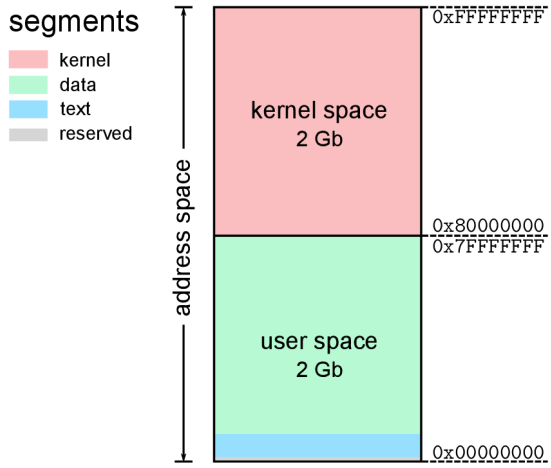

Figura 1

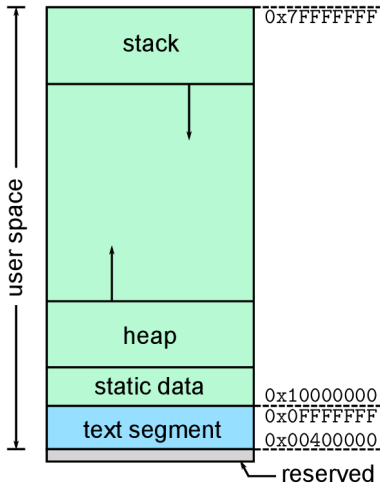

Figura 2

Pero una mirada más detenida de la Figura 2 nos muestra que existe un *heap* al cuál no tenemos acceso aún. Para ello debemos utilizar el *system call* **sbrk**. Esta llamada al sistema nos devuelve la dirección de memoria en $v0 de un bloque de tamaño $a0 solicitado. Es claro a priori que *sbrk* tiene relación con *malloc* en C. Por eso llamamos tambien al *heap* como memoria dinámica.

En MIPS R2000 tanto el heap como el stack comparten el espacio de direcciona-miento. Por eso el stack crece hacia direcciones decrecientes mientras que el heap lo hace en direcciones crecientes en el sentido contrario despues del área de datos estáticos. Eso le permite utilizar la región del segmento de datos en forma óptima como se observa en la Figura 2.

Un estudio pormenorizado de malloc queda fuera de discusión, pero está claro que su implementación en MIPS solicita bloques al sistema operativo con sbrk llevando con algún método la contabilidad de los mismos y utiliza *free* para liberar partes que puedan ser reutilizada por el mismo programa. Es importante entender que dicha liberación es local al ámbito de ejecución del programa en curso y que en caso de MIPS no hay una syscall para devolver memoria. El espacio solicitado se libera al hacer *exit*.

La observación anterior conlleva a la dolorosa conclusión que en assembly la gestión de memoria dinámica debe hacerse con el propio programa o con una librería creada a tal fin que no disponemos.

Lista enlazada simple

Supongamos que queremos almacenar una lista enlazada de números enteros. Cada nodo de la lista contendría dos palabras, el entero y la dirección al próximo nodo. Se asume que en el data segment hemos declarado un puntero al primer nodo llamado *slist* cuyo contenido inicial es *null*. Por lo tanto una función *newnode* para anexar un nuevo número entero en $a0 a la lista podría tener la siguiente forma:

```
# void newnode(int number)
newnode: move $t0, $a0        # preserva arg 1
         li   $v0, 9
         li   $a0, 8
         syscall              # sbrk 8 bytes long
         sw   $t0, ($v0)      # guarda el arg en new node
         lw   $t1, slist
         beq  $t1, $0, first  # ? si la lista es vacia
         sw   $t1, 4($v0)     # inserta new node por el frente
         sw   $v0, slist      # actualiza la lista
         jr   $ra
first:   sw   $0, 4($v0)      # primer nodo inicializado a null
         sw   $v0, slist      # apunta la lista a new node
         jr $ra
```

Ahora ejecutemos nuestra subrutina leaf con la siguiente lista de números en este orden: 1, 2, 3, asumiendo que *sbrk* devuelve la siguiente dirección de memoria 0x10040000 en $v0 en la primer llamada y que el puntero *slist* esta inicializado como corresponde en la dirección 0x10001000 como se muestra en el código a continuación.

```
         .data 0x10001000
slist:   .word 0              # inicializado a null
numbers: .word 1,2,3          # lista de enteros

         .text
main:    la   $s0, numbers
         li   $s1, 3
loop:    lw   $a0, ($s0)
         jal  newnode
         addi $s0, $s0, 4
         addi $s1, $s1, -1
         bnez $s1, loop
         .end
```

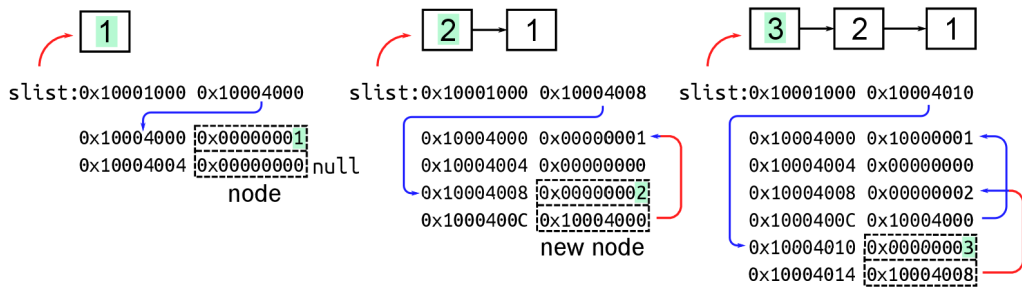

Figura 3 a) b) c)

Luego de la primer ejecución del bucle *loop* el nodo tendría la forma que se esboza en la Figura 3a. En las siguiente figuras se observa el estado de la lista con sus

direcciones de memoria luego de la segunda y tercera llamada. En la mayoría de los casos los nodos toman posiciones consecutivas pero esto no es garantizado[1].

Actividad propuesta

Se solicita que realice un programa en assembly de MIPS R2000 que maneje listas de objetos en forma categorizada utilizando listas enlazadas dobles circulares como se propone en la Figura 4.

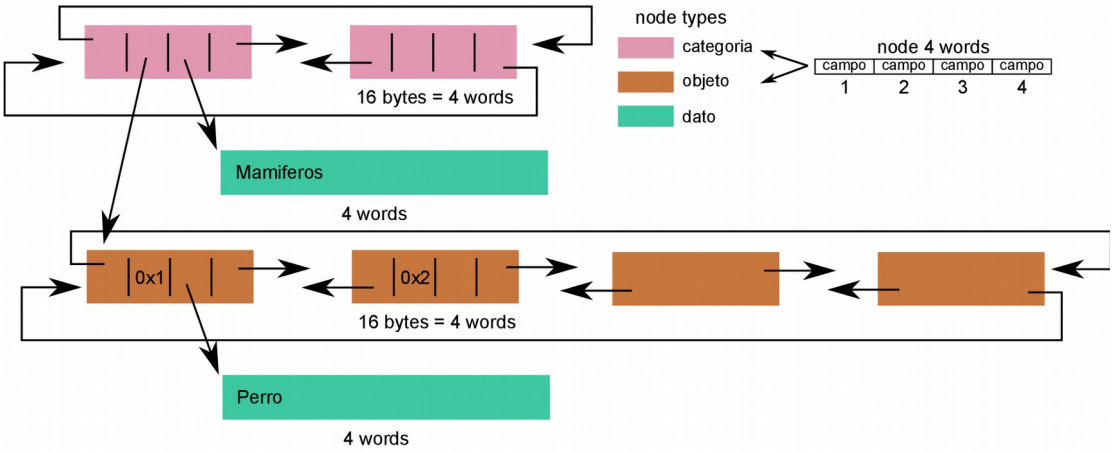

Figura 4

Todos los **bloques** son de 4 palabras para simplificar la administración de memoria. Los nodos de la lista de categorías son muy similares a los nodos de la lista de objetos, ya que su primer campo es un puntero a su antecesor, el cuarto campo otro puntero a su sucesor y el tercer campo un puntero a un bloque de datos tipo string terminados en null que tiene el mismo tamaño que un nodo. En lo que difieren es que los nodos de categoria utilizan el segundo campo para apuntar a otra lista enlazada doble circular.

Cuando un bloque de memoria se elimina lo que se hace es pasarlo a una lista enlazada simple como la mencionada, que sería similar a *free* en C. Cuando se necesita un bloque nuevo se debe examinar si hay algun bloque en la lista de liberados y tomarlo de allí. Si la lista está vacía se recurre al syscall *sbrk*. Este comportamiento es muy similar a *malloc* en C pero muchísimo más siemple. Cuando intente programar este comportamiento entenderá la enorme complejidad de *malloc* que permite solicitar bloques de memoria de tamaño arbitrario. Tambien puede entender que al liberar memoria y solicitar memoria dentro de una aplicación con solicitudes de diferentes tamaños hay muchos espacios de memoria que no podrán utilizarse por ser muy pequeños y que la memoria quedará como un queso gruyere, a esto se le denomina *memory fragmentation* o framentación de la memoria. Cuando un docente le informe que el lenguaje tiene un *garbage collector*, se refiere que tiene todo una gestión compleja para administrar la memoria como Java y no es necesario realizar todo este trabajo y contemplar todos estos problemas, por supuesto toda ventaja tiene su desventaja.

La aplicación tendrá un menú conteniendo todas las actividades que se pueden realizar sobre dichas listas. A continuación se enuncia los items que debe contener el mismo explicando que acción realizan con detalle:

[1]Estamos en un simulador monotarea y no hay competencia por los recursos.

1. Crear una nueva categoría. Si es la primer categoría se considera que es la categoría seleccionada en curso. Si no lo es, se agrega siempre por detras y no cambia la categoría en curso.
2. Seleccionar una categoría. Esto se hace con dos opciones en el menú: pasar a la categoría siguiente o a la anterior respecto a la actual. Si no hay categorías se informará el error (201), si hay una sola categoría se informará el error (202). En cualquier otro caso se informará cuál categoría se ha seleccionado.
3. Listar las categorias. Se aconseja debido a lo primitivo de la consola mostrar con un símbolo “>” la categoría seleccionada en curso. Si no hay categorías se informará el error (301).
4. Borrar una categoría seleccionada. Este comportamiento es complejo porque puede haber muchos nodos de objetos enlazados a la misma. Si se invoca cuando no hay categorías debe informar el error (401). Si la lista de objetos de la categoría seleccionada está vacía entonces debe borrarse solo la categoría y seleccionar como categoría automáticamente la siguiente si existe, de lo contrario deberá nulificar los punteros necesarios. Si en cambio no está vacía deberá primero proceder a borrar todos los objetos devolviendo la memoria correspondiente y luego proceder a borrar la categoría con las mismas salvedades que se indicaron.
5. Anexar un objeto a la categoría seleccionada en curso. Se usará como ID la autonumeración local a cada lista, esto quiere decir que el nuevo ID es siempre uno mayor que el último de la lista de la categoría en curso seleccionada. Si la lista está vacía el ID default es 1. Si se invoca cuando no hay categorías debe informar el error (501).
6. Listar todos los objetos de la categoría en curso. Si no hay objetos para la categoría en curso se informará el error (602) y si no hay categorías creadas se informará el error (601).
7. Borrar un objeto de la categoría seleccionada en curso usando el ID. Si el ID provisto no es encontrado se informará con un mensaje `notFound` y si no existen categorías el error (701).

Debido a las pocas capacidades de la consola del simulador ***mars*** se recomienda utilizar un sistema de menus basado en números y redibujar el menu todas las veces que lo considere necesario, ignorar los acentos y cualquier símbolo que no esté dentro de los caracteres usuales de la tabla ASCII. En caso de solicitar una opción en un contexto incorrecto, por ejemplo listar categorías cuando no ha creado ninguna anteriormente, deberá imprimir un mensaje de error numérico como se explicó anteriormente en detalle. Si el error fuera una selección inexistente del menú, el códido de error sería (101). Si la operación modifica el estado del sistema correctamente se informará con un mensaje que se realizó con éxito y se reimprimirá el menu solicitando otra opción al menos que se seleccione cerrar la aplicación.

Arquitectura de la aplicación

A partir de los objetivos se puede modularizar el código de la siguiente forma: crear las primitivas necesarias para manipular ambas listas circulares, respetando las dependencias entre ellas, tal como lo exije el comportamiento de la aplicación. En el caso de las categorías tendríamos *newcaterogy*, *nextcategory*, *prevcategory*, *listcategories* y *delcategory* para cumplir con los primeros 4 puntos. Para los objetos tendríamos *newobject*, *listobjects* y *delobject*. Las mismas no tienen argumento pero devolverán un entero positivo indicando el error o cero.

Adicionalmente a los mensajes declarados en el segmento de datos *static* (`data`), se reservará espacio para tres punteros *slist*, *cclist*, *wclist* (*circular category list and working category list*) y un vector de direcciones *schev* (*scheduler vector*) de tamaño 32 bytes, debido a las 8 opciones del menu. El primer puntero lo utilizan las funciones *smalloc* y *sfree* que se presentarán a continuación. El puntero *cclist* apunta a la lista de categorías y *wclist* a la categoría seleccionada en curso. El vector schev contiene las direcciones de todas las funciones que debe programar con las etiquetas mencionadas arriba. Al iniciar la aplicación todos deberán estar correctamente apuntados a NULL y el vector deberá cargarse con dichas direcciones. A continuación se provee el segmento para que todos los trabajos presenten una interfaz uniforme.

```
                .data
slist:          .word 0
cclist:         .word 0
wclist:         .word 0
schedv:         .space 32
menu:           .ascii  "Colecciones de objetos categorizados\n"
                .ascii  "====================================\n"
                .ascii  "1-Nueva categoria\n"
                .ascii  "2-Siguiente categoria\n"
                .ascii  "3-Categoria anterior\n"
                .ascii  "4-Listar categorias\n"
                .ascii  "5-Borrar categoria actual\n"
                .ascii  "6-Anexar objeto a la categoria actual\n"
                .ascii  "7-Listar objetos de la categoria\n"
                .ascii  "8-Borrar objeto de la categoria\n"
                .ascii  "0-Salir\n"
                .asciiz "Ingrese la opcion deseada: "
error:          .asciiz "Error: "
return:         .asciiz "\n"
catName:        .asciiz "\nIngrese el nombre de una categoria: "
selCat:         .asciiz "\nSe ha seleccionado la categoria:"
idObj:          .asciiz "\nIngrese el ID del objeto a eliminar: "
objName:        .asciiz "\nIngrese el nombre de un objeto: "
success:        .asciiz "La operación se realizo con exito\n\n"
```

Para inicializar *schev* mostramos un código a modo de ejemplo para las primeras dos funciones que debe colocarse al principio de *main* como se muestra a continuación:

```
main:   la      $t0, schedv         # initialization scheduler vector
        la      $t1, newcaterogy
        sw      $t1, 0($t0)
        la      $t1, nextcategory
        sw      $t1, 4($t0)
        ...
```

A continuación presentamos dos funciones *smalloc* y *sfree* cuyo objetivo es realizar la gestión de memoria utilizando la *system call* **sbrk** cuando corresponda. Estos tienen prototipos muy sencillos: *node* smalloc()* y *void sfree(node*)*. Como todos los

bloques son de 4 palabras su comportamiento es muy sencillo tal como fué descripto en el punto lista enlazada simple. Puede adaptarse facilmente *newnode* para este propósito, como se muestra a continuación:

```
smalloc:
        lw      $t0, slist
        beqz    $t0, sbrk
        move    $v0, $t0
        lw      $t0, 12($t0)
        sw      $t0, slist
        jr      $ra
sbrk:
        li      $a0, 16     # node size fixed 4 words
        li      $v0, 9
        syscall             # return node address in v0
        jr      $ra

sfree:
        lw      $t0, slist
        sw      $t0, 12($a0)
        sw      $a0, slist  # $a0 node address in unused list
        jr      $ra
```

En la aplicación usted no hará uso de las mismas, ya que la cátedra considerando la complejidad de abordar el desarrollo de esta aplicación le proveerá de tres funciones que realizan la gestión de los nodos y bloques, que son las que la utilizan. Sin embargo debe entender claramente su funcionamiento para poder aborar el debuging de su aplicación.

Dada la similitud que existe entre nodos, como se observa en la figura 4, se puede generalizarse el manejo de los mismos mediante *node* addnode(list, node*)* y *delnode(node*, list)*. La primera devuelve un nodo nuevo *node** agregado al final de la lista y si *list* es NULL se asume que es nueva. Para borrar un nodo se provee su dirección en *node* y la dirección de la lista en *list*.

Adicionalmente se provee *block* getblock(char *)* que solicita un string usando el mensaje provisto por *char ** y devuelve un puntero *block** al bloque de memoria conteniendo dicho string.

El fuente de todas ellas está disponible en el anexo. Para su mejor entendimiento se provee como ejemplo de uso el código fuente de la función *newcategory*:

```
newcaterogy:
        addiu   $sp, $sp, -4
        sw      $ra, 4($sp)
        la      $a0, catName            # input category name
        jal     getblock
        move    $a2, $v0                # $a2 = *char to category name
        la      $a0, cclist             # $a0 = list
        li      $a1, 0                  # $a1 = NULL
        jal     addnode
        lw      $t0, wclist
        bnez    $t0, newcategory_end
        sw      $v0, wclist             # update working list if was NULL
newcategory_end:
        li      $v0, 0                  # return success
        lw      $ra, 4($sp)
        addiu   $sp, $sp, 4
        jr      $ra
```

Anexos

```
        # a0: list address
        # a1: NULL if category, node address if object
        # v0: node address added
addnode:
        addi    $sp, $sp, -8
        sw      $ra, 8($sp)
        sw      $a0, 4($sp)
        jal     smalloc
        sw      $a1, 4($v0)             # set node content
        sw      $a2, 8($v0)
        lw      $a0, 4($sp)
        lw      $t0, ($a0)              # first node address
        beqz    $t0, addnode_empty_list
addnode_to_end:
        lw      $t1, ($t0)              # last node address
        # update prev and next pointers of new node
        sw      $t1, 0($v0)
        sw      $t0, 12($v0)
        # update prev and first node to new node
        sw      $v0, 12($t1)
        sw      $v0, 0($t0)
        j       addnode_exit
addnode_empty_list:
        sw      $v0, ($a0)
        sw      $v0, 0($v0)
        sw      $v0, 12($v0)
addnode_exit:
        lw      $ra, 8($sp)
        addi    $sp, $sp, 8
        jr      $ra

        # a0: node address to delete
        # a1: list address where node is deleted
delnode:
        addi    $sp, $sp, -8
        sw      $ra, 8($sp)
        sw      $a0, 4($sp)
        lw      $a0, 8($a0)             # get block address
        jal     sfree                   # free block
        lw      $a0, 4($sp)             # restore argument a0
        lw      $t0, 12($a0)             # get address to next node of a0
node
        beq     $a0, $t0, delnode_point_self
        lw      $t1, 0($a0)             # get address to prev node
        sw      $t1, 0($t0)
        sw      $t0, 12($t1)
         lw       $t1, 0($a1)                # get address to first node
again
        bne     $a0, $t1, delnode_exit
        sw      $t0, ($a1)              # list point to next node
        j       delnode_exit
delnode_point_self:
        sw      $zero, ($a1)            # only one node
delnode_exit:
        jal     sfree
        lw      $ra, 8($sp)
        addi    $sp, $sp, 8
        jr      $ra
```

```
        # a0: msg to ask
        # v0: block address allocated with string
getblock:
        addi    $sp, $sp, -4
        sw      $ra, 4($sp)
        li      $v0, 4
        syscall
        jal     smalloc
        move    $a0, $v0
        li      $a1, 16
        li      $v0, 8
        syscall
        move    $v0, $a0
        lw      $ra, 4($sp)
        addi    $sp, $sp, 4
        jr      $ra
```