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**.

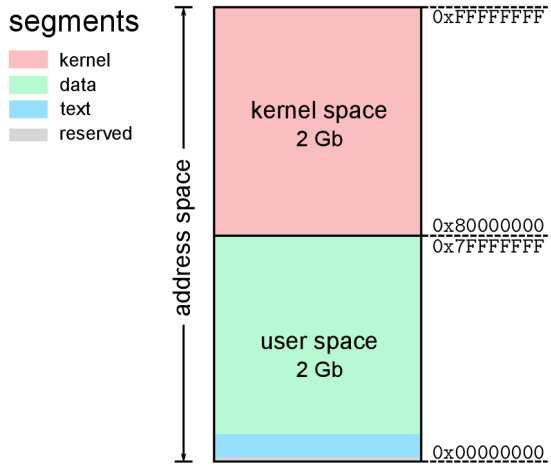

Figura 1

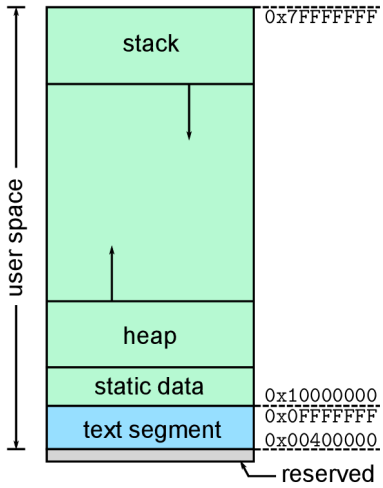

Figura 2

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
```

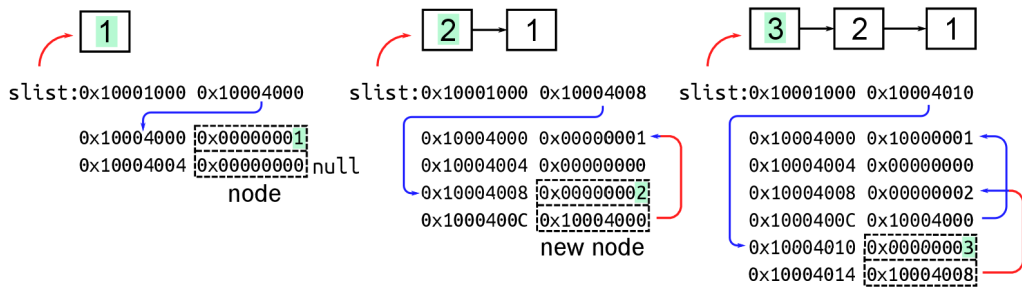

Figura 3 a) b) c)

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.

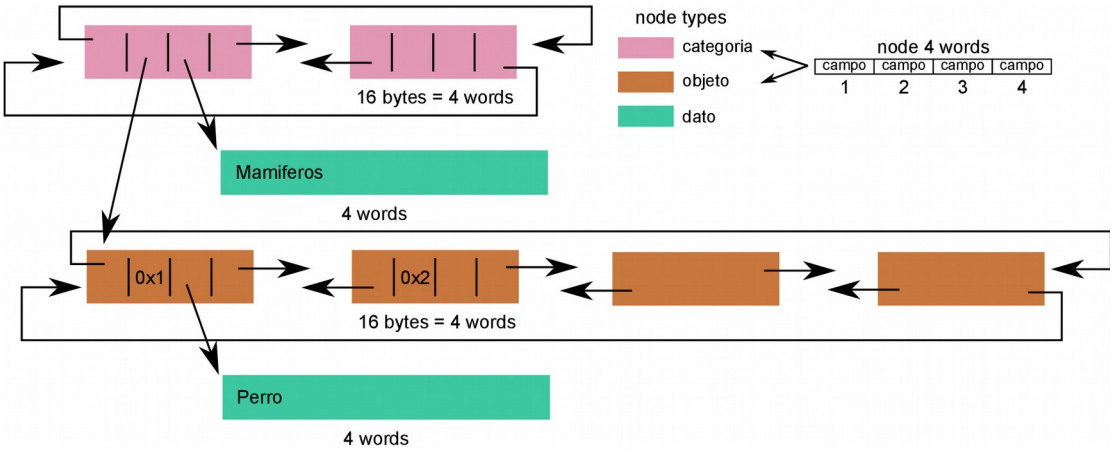

Figura 4

Todos los nodos 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 tambien son nodos. En lo que difieren es que los nodos de categoria utilizan el segundo campo para apuntar a otra lista enlazada doble circular.

Cuando un nodo 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 nodo nuevo se debe examinar si hay algun nodo 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.

Arquitectura de la aplicación

Lo primero que debierá realizar son dos funciones *smalloc* y *sfree* que 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 párrafo anterior. Puede adaptarse facilmente *newnode* para este propósito, como se muestra a continuación:

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

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

Adicionalmente deberá considerar en el segmento de datos *static* el espacio para los punteros *slist*, *cclist*, *wclist*, buffer de ingreso de strings (optativo) y mensajes. El primer puntero ha sido explicado y el siguiente apunta a la lista de categorías y a la seleccionada en curso (*circular category list and working category list*).

Luego deberá tener las primitivas necesarias para manipular ambas listas circulares. En el caso de las categorías tendríamos *newcaterogy*, *nextcategory*, *prevcategory* y *delcategory*. Para los objetos tendríamos *newobject* y *delobject*. Dada la similitud de sus nodos puede generalizarse el manejo de los mismos con las primitivas *node* addnode()* y *delnode(node*)*.

Se recomienda pensar el problema primero en C (opcional) y modularizar el comportamiento lo mejor posible.

A continuación se enuncia una serie de objetivos que debe cumplir el software:

1. Crear una categoría vacia. Si es la primer categoría se considera que es la categoría seleccionada 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.
3. Listar las categorias. Se aconseja debido a lo primitivo de la consola mostrar con un * la categoría seleccionada en curso.
4. Borrar una categoría seleccionada. Este comportamiento es complejo porque puede haber muchos nodos de objetos enlazados a la misma.
5. Anexar un objeto a la categoría seleccionada en curso. Se uso como ID la autonumeración local, 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.
6. Borrar un objeto de la categoría seleccionada en curso usando el ID.
7. Listar todos los objetos de la categoría en curso.

Se aconseja nuevamente debido a las pocas capacidades de la consola del simulador *spim* utilizar un sistema de menus basado en números y redibujar el menu todas las

veces que lo considere necesario[2], 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, deberá imprimir un mensaje aclaratorio de error.

Bonus

Un problema relevante para cualquier aplicación de este tipo es cargar los datos en cada prueba. Si la aplicación debe cerrarse, todo este trabajo se pierde. Decimos que la aplicación es *stateless*.

Cuando buscamos persistencia entre ejecuciones queremos que la aplicación pueda salvar su estado, por ejemplo mediante un volcado de memoria o *memory dump*. Otra alternativa es modelar de alguna forma las listas independiente de los espacios de memoria y volcar las mismas en dicho formato al disco. Esto permite independizarse tanto de la futura ubicación del heap como de la fragmentación interna. Por ejemplo un formato ASCII pasible puede ser:

```
category:object,object,...
```

La cátedra le encomienda en forma totalmente **opcional**, que investigue el tema de acceso al disco y provea un mecanismo para almacenar el estado de su programa al salir y leerlo antes de comenzar el resto de una nueva ejecución.

[2]La cátedra reconoce que este trabajo representa una tarea ardua, por eso solicitamos que trabajen en grupos de 5 a 7 alumnos o más.