UPR-Mayaguez

ICOM 4035 – PROYECTO 1

Segundo Semestre 2010-2011[1]

Nota: Éstas son las especificaciones de la primera etapa de un proyecto de programación. En ésta va a trabajar con una parte limitada del mismo, aunque esencial para el resto. Tiene que leer esto rápido, pero con el detenimiento suficiente para entender claramente lo que se especifica. En caso de encontrar errores o inconsistencias, favor de informar las mismas tan pronto le sea posible. De tener alguna duda con alguna parte, las mismas se atenderán durante las horas de oficina.

1. Introducción

En este proyecto vamos a trabajar con un sistema que manejará un recurso de memoria a usarse para almacenar datos de distintos tipos y acceder a los mismos. El sistema final podrá responder a distintas operaciones que usuarios del mismo van a poder solicitar a través de comandos. Para eso, en fases futuras vamos a definir una serie de comandos que van a permitir distintas operaciones que en gran parte van a ser semejantes a operaciones que las que se llevan a cabo en programas escritos en lenguajes como Java: manejo de variables de distintos tipos de dato, *reservar espacio* en memoria, *guardar valores* en áreas reservadas de memoria, *acceder a datos* que estén almacenados en la memoria y *liberar espacio* en áreas de memoria que no se necesiten, etc.

Ustedes recibirán una clase que provee el funcionamiento de la *memoria,* tal y como lo vamos a requerir en este proyecto. Esa clase (que llamaremos **`Memory`**) implementa todas las operaciones que vamos a tener para el componente del sistema que representa su memoria. En esta etapa vamos a trabajar con la implementación de algunas estructuras de datos adicionales que se van a necesitar para eventualmente lograr la funcionalidad del sistema que se persigue. La versión del sistema que resulte al concluir esta primera etapa va a proveer operaciones que son necesarias para poder hacer al menos lo siguiente: *reservar y asignar espacio de* memoria a variables u objetos, para *liberar espacio en memoria*, para *copiar secuencia de bytes* (bloques de bytes) *de un área de memoria a otro área* de la memoria.

En fases futuras vamos a incorporar el manejo de distintos *tipos de datos*, cuyos valores posibles vamos a poder almacenar en la memoria y accederlos usando *variables*. Éstos son los siguientes (puede ser que luego incluyamos otros): **`int`**, **`boolean`**, **`char`**, **`array`** y *referencia a objeto*. Vamos a tener las siguientes restricciones en cuanto a la cantidad de memoria que requiere cada tipo de datos con los que se trabaja en este proyecto. Un valor tipo **`int`** ocupa 4 bytes de memoria, un valor tipo **`char`** ocupa 1 byte (asumimos solamente código ASCII), un valor tipo **`boolean`** ocupa 1 byte. Un valor tipo **`array`** es una referencia a un objeto que se comportará como un arreglo de una dimensión en lenguajes como Java. Hablaremos sobre esto más adelante. Un valor tipo *referencia a un objeto* es un número entero no negativo y ocupa 4 bytes.

El sistema final que se persigue en este proyecto (no en esta primera fase) va a poder procesar una serie de comandos simples (dados en forma de texto – no interfases gráficas). Las siguientes son algunas de las operaciones que esperamos poder lograr eventualmente a través de esos comandos.

1. Declarar una variable de tipo **`int`**, de tipo **`boolean`**, de tipo **`char`**, de tipo **`array`**, o de tipo *referencia a un objeto*. Una declaración reserva espacio en memoria para la variable declarada y del tamaño apropiado. En el caso de una variable de referencia (una variable tipo **`array`** o de algún otro tipo no primitivo), lo que se reserva es espacio de 4 bytes para el lugar en memoria donde se almacena la referencia de un objeto, que ocupa otra área en memoria, y cuya dirección es esa referencia. Esto debe funcionar de manera similar a como funciona en Java.
2. Definir un tipo de objeto no primitivo.

[1] Prof. Pedro I. Rivera Vega, ECE Department, UPR-Mayaguez, p.rivera@upr.edu

3. Asignar valor a una variable.
4. Acceder el valor de una variable.
5. Mostrar valor de una variable.
6. Hacer garbage collection.
7. Compactar la memoria usada.
8. Mostrar contenido de la memoria en una región dada.

A continuación describimos el componente de la memoria que se va a estar usando. Luego describimos en detalle lo que se tiene que hacer en esta primera etapa del proyecto.

2. Componente de Memoria y Su Manejo

La memoria que se va a manejar es una en la que *cada celda es de tamaño igual a* `1` *byte*. Cada celda (o byte) tiene una dirección única en la memoria, la cual es un número entero positivo que va desde **`0`** hasta **`N-1`**, donde **`N`** es el número de bytes en la memoria (o el tamaño o capacidad de la memoria). Esta memoria maneja *words* de tamaño igual a 4 bytes. Esto quiere decir que las operaciones de escribir a la memoria o de leer de ella siempre trabajan con un "memory word" completo. Y aunque cada byte tiene dirección propia, la memoria solamente permite acceso a áreas cuya dirección (la dirección de su primer byte) es un múltiplo de 4 y cada operación afecta (bien sea porque le escribe algo nuevo encima, o porque los lee) los 4 bytes (el word) que comienzan desde esa dirección inicial. Note que eso no quiere decir que no se pueda acceder a un byte cuya dirección no sea un múltiplo de 4, pero esto habría que hacerlo accesando el word que lo contenga y con algunas operaciones externas a las que provee la memoria. Para los efectos de ésta, las operaciones de leer o escribir en ella siempre trabajan con una palabra (memory word) completa.

Para este trabajo ya se ha implementado una clase que permite la creación de objeto que provee la funcionalidad de la memoria que se describe en el párrafo previo. Dicha clase la llamamos **`Memory`**. Ustedes tienen que estudiar y entender la implementación de esta clase, pero NO PUEDEN MODIFICARLA. He tratado de simular de manera lo más fiel posible (pero evitando complicaciones) la manera como trabajan algunos tipos de memoria.

Operaciones Primitivas de la Memoria

Vamos a hablar algo de las operaciones básicas que se requieren para una memoria. Esas operaciones lo que deben permitir es básicamente lo siguiente:

1. Escribir (grabar) un word en algún lugar en la memoria
2. Leer el contenido de un word en un lugar dado en la memoria

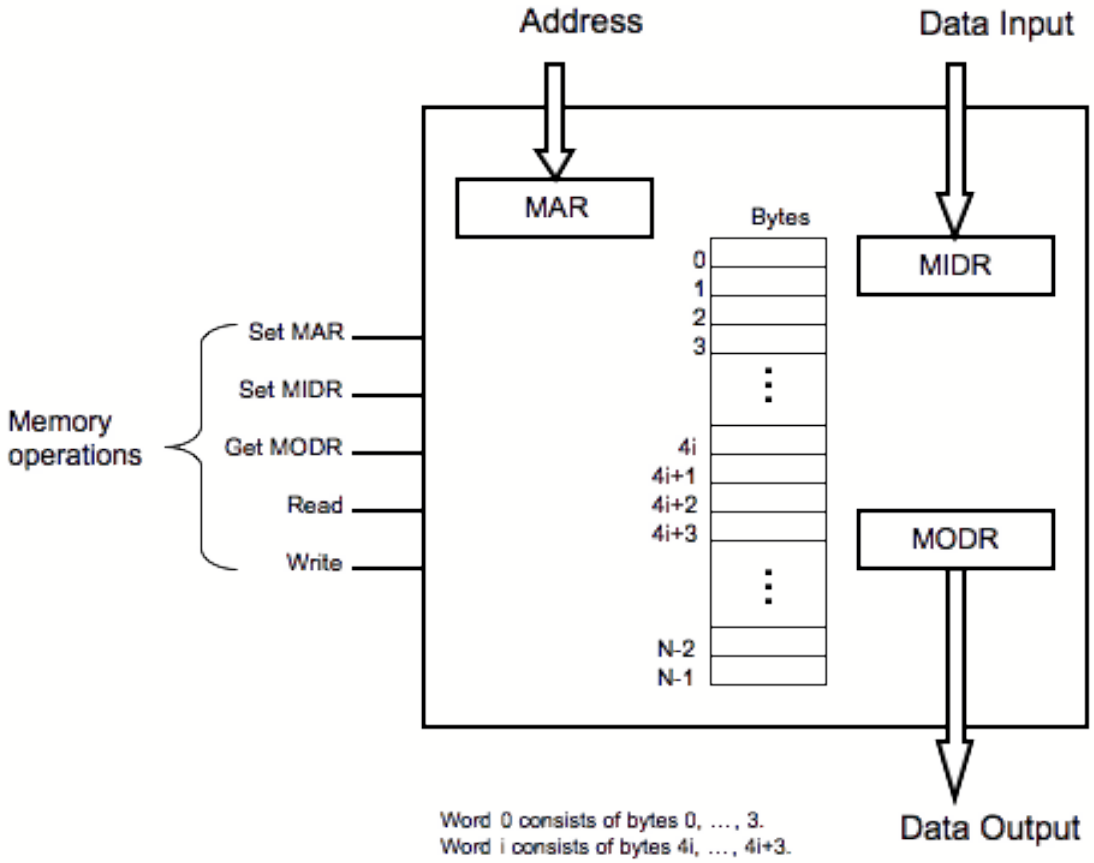
En cada caso el lugar de la memoria que se afectará se identifica con la dirección de su primer byte (la cual, como ya dijimos, tiene que ser un múltiplo de 4). Así que ambas operaciones necesitan recibir la dirección del word al que se va a acceder. La memoria que vamos a estar usando (clase **`Memory`**) consiste de un área de muchos bytes, dos registros de data: "*memory input data register*" (MIDR) y "*memory output data register*" (MODR). Toda operación de leer o escribir necesita la dirección de la palabra de memoria que se lee o sobre la que se escribe. Esa dirección se almacenará en un registro especial en la memoria, el cual denominamos como "*memory address register*" (MAR).

Esta memoria solo acepta 5 operaciones primitivas (sobre éstas podremos luego construir las dos básicas que mencionamos anteriormente: leer y escribir). Éstas son:

1. Guardar un valor entero (que se interpretará como una dirección) en su registro MAR. Vea la operación **`setAddress`**.
2. Copiar el contenido de un registro (4 byte) a su registro para input (MIDR) a la memoria. Vea la operación **`setInput`**.
3. Producir una copia (un registro) de su registro para output (MODR) de la memoria. Vea la operación **`getOutput`**.

4. Grabar el contenido del registro MIDR al área de memoria cuya dirección esté en el registro MAR en el momento. Si la dirección en MAR no es correcta, entonces se generan distintos tipos de excepciones. Ver operación **`write`**.
5. Copiar al registro MODR el contenido del área de memoria (4 bytes) cuya dirección inicial esté en el registro MAR en el momento. Si la dirección en MAR no es correcta, entonces se generan distintos tipos de excepciones. Ver operación **`read`**.

La siguiente figura resume la organización de la memoria considerada.



Un registro es un componente especial en el que se puede almacenar un word. Para manejar los registros que se mencionan también se ha incluido la clase **`Register`** (que tampoco puede modificar). Estudie esta clase y trate de entender bien la manera como se implementan las operaciones. Sólo menciono lo siguiente. Cuando se copia el contenido de un registro a área en la memoria que comienza con dirección **`x`**, donde **`x`** es un múltiplo de 4, el contenido del registro en sus bytes 0, 1, 2 y 3, se copian a los bytes de memoria con direcciones **`x`**, **`x+1`**, **`x+2`** y **`x+3`**, respectivamente. (Piense en cómo debe ser entonces la operación de copiar a un registro el contenido del lugar en memoria cuyo primer byte tiene dirección **`x`**.) El contenido de un registro se puede acceder byte por byte. Cuando el registro contiene un dato que corresponda a un número entero (como un valor tipo **`int`** en Java), su byte 0 contiene la parte menos significativa de dicho valor, mientras que el byte 3 contiene su parte más significativa. Los números se representan usando *representación 2's complement* para un área de 32 bits (4 bytes). Debe buscar información sobre esta manera de representación de números enteros pues es así como se representan los números en la memoria de una computadora real...

Operaciones Sobre el Componente de Memoria

Note que la memoria provee unas operaciones básicas, que no puede modificar (Algunas memorias reales funcionan de manera similar…). Hay que tener en cuenta entonces cómo

usarlas correctamente para lograr que la memoria se pueda utilizar para almacenar y recuperar data correctamente.

Para escribir (grabar) un valor (el contenido de un word - por ahora piense sólo en enteros) en un lugar específico en la memoria en general tiene que llevar a cabo tres operaciones consecutivas con ese componente de memoria. Éstas son:

1. Colocar el contenido a escribir en el registro MIDR de la memoria.
2. Colocar la dirección a dónde lo va a escribir en el registro MAR de la memoria.
3. Aplicar la operación **`write`** a la memoria.

Note que las primeras dos operaciones pueden ejecutarse en cualquier orden. (Estudie las operaciones correspondientes que se han implementado en la clase **`Memory`**.)

Usted debe poder derivar, de manera similar, las tres operaciones que se necesitan para acceder al valor que se tenga en una dirección específica en la memoria.

Ahora, vea que lo que la memoria entiende y sabe hacer son unas tareas simples. En un programa (por ejemplo, en Java) se tienen que manejar variables, reservar espacios para éstas en memoria, asignarle valores a la variable, acceder al valor de la variable, etc. Todo esto tiene que hacerse con esas operaciones primitivas y con la ayuda de algunas estructuras de dato que permiten mantener registro de lo necesario para crear y manejar la abstracción que representan las variables, los tipos de dato, las operaciones de reservar espacio en la memoria, las operaciones de asignación, etc. En esta primera etapa del sistema trabajamos con la reservación de espacio en la memoria y el garantizar que esas reservaciones sean manejadas de manera correcta. Vamos a ver.

Reservación de Espacio en Memoria y su Mannejo

En cada momento en que se necesita reservar espacio en memoria para una variable o un objeto se necesita buscar un área en memoria disponible para poder reservar la cantidad de bytes requeridos para la reservación que se esté intentando. Si no hubiera esa cantidad de bytes disponibles en un bloque contiguo, entonces la operación debe generar una condición de falta de memoria, lo que va a causar que el sistema aborte luego de producir un mensaje apropiado para la situación. Para eso es importante que se lleve un registro de las áreas en memoria que están disponibles en cada momento. Cada vez que se inicia la ejecución del sistema toda la memoria está disponible. Si necesita reservar **`n`** bytes para algún tipo de dato, pues puede reservar los bytes con direcciones **`0`** ... **`n-1`**. Si luego necesita **`k`** bytes, pues entonces puede reservar los bytes con direcciones **`n`** ... **`n+(k-1)`**, y así sucesivamente. Esto parece fácil, y lo sería si durante la ejecución del sistema no se liberaran áreas de memoria una vez se reserven, o si todas las áreas de memoria que se reservan fuesen siempre del mismo tamaño. Pero el que se puedan liberar áreas reservadas (por que ya no se requieren), acompañado de la variedad posible en sus tamaños genera *fragmentación de la memoria*; esto es, van a haber lugares disponibles entre lugares reservados.

Un problema con la fragmentación es que en ocasiones se puede requerir un espacio de memoria que, aunque en la suma total de la memoria disponible haya suficiente para cumplir con el requerimiento, el mismo no se pueda satisfacer porque no hay un bloque con tantos bytes contiguos que estén disponibles. Una solución es compactar la memoria. Esto lo que hace es mover todos las áreas de memoria a uno de los extremos de la memoria, usualmente al inicio de ella. La refragmentación es una operación lenta. Requiere mover todos los datos y ajustar las referencias necesarias para que, desde el punto de vista de las variables del momento y sus valores, etc., no se perciban cambios.

En este proyecto se va a requerir que se mantenga el control necesario para que la memoria sea utilizada al máximo, haciendo posible la refragmentación, la localización de espacio que cumpla con un pedido, la contabilidad de qué bytes están ocupados y cuáles están disponibles.

Una manera simple de mantener el registro de bytes disponibles es usando una lista ligada de todos los bloques de memoria que están disponibles en el momento. A la misma se le tiene que dar mantenimiento para que en todo momento refleje de manera fiel el estado actual de la memoria según se van reservando o liberando espacios. El contenido de la lista es simple, cada elemento es un par de la forma **`(n1, n2)`**, que indicará que el bloque de bytes, con direcciones desde **`n1`** hasta **`n2`**, están disponibles. En cada caso, el par indica un bloque maximal, esto es (si **`n1`** no es **`0`**), el byte con dirección **`n1-1`** está reservado, y, si **`n2`** no es el byte con dirección más alta en la memoria, entonces **`n2+1`** está reservado. Si vemos cada par como un intervalo cerrado, esto también implica que pares distintos en la lista no pueden intersecarse. Los elementos en la lista se mantienen ordenados por el primer componente de cada par.

Veamos un ejemplo. Suponga que la memoria tiene capacidad de 20 bytes, con direcciones 0 .. 19. Si la lista contiene los pares (4, 7), (10, 13) y (15, 19), entonces los lugares de la memoria que están reservados son los bytes con direcciones 0, 1, 2, 3, 8, 9 y 14, mientras que el resto está disponible. Si ahora se liberan los bytes 2 y 3, entonces la lista tiene que modificarse para que refleje este cambio. En este caso, la nueva lista será: (2, 7), (10, 13) y (15, 19). Si ahora se liberan los bytes 4 al 6, entonces la nueva lista será: (2, 3), (7, 7), (10, 13) y (15, 19). Esto debe sugerir algunas de las operaciones que van a ser necesarias con los elementos de la lista cada vez que se reservan o liberan áreas de memoria.

3. Lo que Tiene que Hacer en Esta Fase 1

En esta fase del proyecto lo que va a hacer es trabajar con una clase que cumpla con proveer operaciones para:

- **reservar áreas de memoria de un tamaño específico** – En todos los bytes reservados se almacenará el caracter '1'.
- **liberar áreas en memoria de tamaño que se especifica y comenzando en una dirección que se especifica** – En todos los bytes disponibles se almacenará caracter '0'.
- **llevar a cabo la operación de mostrar el contenido de la memoria en una región dada, byte por byte** (the ASCII character)

Esto se va a hacer con una clase que implementa la siguiente interfase:

```
public interface MemorySystem {
   /** Trata de reservar bloque de memoria que consiste de
     * tantos bytes como se especifica. Esos bytes tienen
     * que estar disponible en el momento.
     * If successful, all reserved byte are initialized
     *  to contain the char value '1'.
     * @param size the number of contiguous bytes to reserve
     * @throws FullMemoryException if memory has no free
     * left.
     * @return a positive integer value corresponding of the
     * address of the first byte in the block that has been
     * reserved. This would mean that there was at least one
     * contiguous block of "size" bytes that were free.
     * those bytes will then be recorded as reserved.
     * If no such block of contiguous free blocks is found,
     * the value to return is -1 and no further action is
     * done by this method.
    **/
   int reserveMemory(int size)
         throws FullMemoryException;
```

```
    /** Tries to free the block of memory of the size given
      * at the address given. If posible, the block of bytes
      * is freed and properly recorded. If not successful,
      * it prints proper error message.
      * If successful, all bytes that were freed are assigned
      * the char value '0'.
     **/
    void freeMemory(int address, int size);

    /** Displays the content of memory from a given initial
      * address up to a final address. The content must show
      * the addres of the byte and its content as a
      * a character – it shows the char value of the content
      * treated as an ASCII code.
      * @param initialAddress the address of the first byte
      * @param finalAddress the address of the last byte
      * PRE: 0 <= initialAddress <= finalAddress < memory_size.
     **/
    void showMMASCII(int initialAddress, int finalAddress);
}
```

El objetivo de cada operación en la interfase previa debe estar claro en base a los comentarios que las describen y a los nombres usados para cada una, pero no trabaje con dudas! En caso de tenerlas, haga las preguntas necesarias…

Su implementación tiene que incluir al menos una clase que llamará **`P1MemSystem`**, la cual implementará esta interface. La misma debe tener al menos lo siguiente:

```
public class P1MemSystem implements MemorySystem {
    private static final int DEFMEMSIZE = 64;
    private Memory mem;     // the memory
    private MemControlStructure mcs;   //to manage free space
    … other internal fields …
    public P1MemSystem() {
       this(DEFMEMSIZE);
       … default initialization of other internal fields…
    }
    public P1MemSystem(int msize) {
       if (msize <= 0)
          msize = DEFMEMSIZE;
       mem = new Memory(msize);
       mcs = new MemControlStructure(msize);
       … initialization of other internal fields…
    }
    … implementation of other required methods …
}
```

La implementación final va a requerir de otras clases. Por ejemplo, la clase que corresponde al tipo de la lista de control que se va a usar para mantener contabilidad de lugares disponibles en la memoria. Esa clase en particular se va a requerir que esté implementada sobre una implementación del ADT **`LinkedList<E>`** (la cual se especificó para ejercicio de laboratorio hace al menos dos semanas) como una lista doblemente ligada con dummy header y dummy trailer nodes (**`DLDHDTList<E>`**). La clase final que representa el tipo de la estructura de control para el manejo de los espacios disponibles en memoria tiene que implementar la siguiente interface:

```
public interface MemControlStructure {
    /**
      * Searches for an available block of contiguous
      * bytes of the given size (number of bytes) according
      * to the current content of the structure.
      * If successful, the structure is modified accordingly
      * to record the space reservation.
      * @param size number of bytes to reserve.
      * @throws InvalidParameterException whenever the
      *       specified size is either negative or exceeds
      *       the size of the memory being manager.
      * @return -1 if not successful (no block of
      *       available contiguous bytes of the requested
      *       size could be found; otherwise, it returns
      *       the address of the first byte in the block
      *       that was reserved.
    **/
    int reserveMemSpace(int size)
         throws InvalidParameterException;

    /**
      * Records the availability of a block of bytes
      * for future use. It basically marks as "available"
      * the specified number of bytes beginning from
      * the byte at the given address.
      * @param address the address of the first byte to free
      * @param blockSize the number of bytes to free.
      * @throws InvalidAddressException whenever the address
      *  given is not valid.
      * @throws BlockBoundaryViolationException whenever
      *  the suggested block is out of the boundaries of
      *  the memory being managed
      * @throws SpaceNotReservedException whenever some of
      *  the bytes in the specified block are already free.
    **/
    void freeSpace(int address, int blockSize) throws
               InvalidAddressException,
               BlockBoundaryViolationException,
               SpaceNotReservedException;
}
```

Debe tener en cuenta que lo anterior es el tipo de un objeto que se usa para mantener la contabilidad de espacios disponibles en una memoria asociada. De alguna forma ese objeto tiene que tener la información necesaria de dicha memoria asociada. En nuestro caso bastaría con saber la capacidad de la memoria asociada. Así que de alguna forma, los constructores deben proveer para eso… Pero tiene que tener en cuenta que estas operaciones no tienen ningún efecto en la memoria asociada: nada se escribe en ella. Recuerde que el objetivo de este tipo de objeto es usarlo como una estructura externa a la memoria, pero que sirve para mantener la información necesaria sobre los lugares de memoria que están disponibles en todo momento.

4. Lo que ha Recibido La implementación parcial que se incluye contiene varias clases. NO PUEDE HACER CAMBIOS EN NINGUNA DE LAS CLASES RECIBIDAS QUE NO SEAN PARA TESTING. Ustedes van a trabajar a partir de esto para lograr implementar la clase **`P1MemSystem`** que se ha descrito.

Esta memoria trabaja más o menos como ya ha sido explicado.

Tiene que entender esta clase, vea los testers que se incluyen y trate otras pruebas. Pueden haber errores! Así que reporte cualquiera que encuentre o que crea que ha encontrado.

5. Fecha de Entrega La fecha de entrega de lo que se le pide en esta etapa es el **Domingo, 13 de marzo 2011, a las 11:59pm**. Cualquier entrega luego de esa fecha, pero en o antes de Martes, 15 de marzo 2011, a las 11:59pm, tendrá una penalidad del 20%. Cualquier entrega luego de esta segunda fecha se considerará como "trabajo no entregado", por lo que recibiría una nota de 0 en su evaluación. **Más adelante se les estarán enviando detalles de cómo es que tiene que hacer su entrega**, pero se espera que la misma sea a través de la cuenta de repositorio que cada cual tiene (o va a tener en los próximos días).

Estén pendientes a sus emails, pueden haber modificaciones a lo previo o extensiones. Quizás para corregir posibles errores encontrados, para aclarar algo, o para agregar algo.