Assignment 2
Due by 23:55 Sunday July 21

Overview

The JVM implementation used in Assignment 1 has a major deficiency – it does not include a garbage collector. Your task is to provide it.

The Current State of the MyJVM System

The supplied code makes use of two sets of storage allocation functions. One set is used for allocating objects and arrays (which are just another kind of object) on the heap, the other set is used for everything else. All the functions are held in one source file: **`MyAlloc.c`**.

Allocation of Storage for Heap Instances

The functions used for managing the storage of heap instances are these:

- **`void InitMyAlloc( int HeapSize )`**

  This function allocates a large block of memory to be used for the Java heap. The default size is 10KB, a size which can be overridden by the **`-H`** command-line option. It is called from **main.c**.

- **`void PrintHeapUsageStatistics()`**

  This function is called at the end of execution from **`main.c`**. It reports some statistics about the number and size of allocated blocks, plus statistics about garbage collections (if there were any being performed).

- **`void *MyHeapAlloc( int size )`**

  This function returns a block of memory in the Java with sufficient space for a class instance or array. It is invoked when these JVM ops are executed: **`ldc`** (for **`String`** constants), **`anewarray`**, **`new`**, and **`newarray`**. It is also invoked when a class is loaded, allocating storage for an instance of the **`Class`** construct.

  The operation of this function is explained in more detail below.

  The command-line option **`-Th`** causes this function to output tracing messages.

- **`static void MyHeapFree( void *p )`**

  There are currently no calls to this function anywhere in the MyJVM system. One of the goals of this assignment is for you to create calls to this function, passing it the addresses of objects on the heap which are known to be garbage.

- **`void gc()`**

  This is the garbage collector function. It is invoked when **`MyHeapAlloc`** cannot satisfy a request to return a block of storage or when the Java program executes a call to **`System.gc()`**.

  The function is currently unimplemented. If called, it does nothing but print an error message and then returns to the caller.

Allocation of Storage for Everything Else

The functions used for general storage allocation are simply wrappers around the standard functions provided in the C library. These wrappers perform some simple checks to make sure that requests for storage do not return a **`NULL`** result, and to make sure that a pointer passed to the free function is plausible (i.e. the address is within the range of memory being managed by the C library functions).

Further Details for JVM Heap Management

When the JVM starts, it calls the **`InitMyAlloc`** function to allocate a single block of storage for the JVM heap.

Blocks of storage allocated in the JVM heap are always multiples of 4 bytes in size and are prefixed by a word which contains the size of the block (including the 4 byte prefix word). Figure 1 illustrates this.

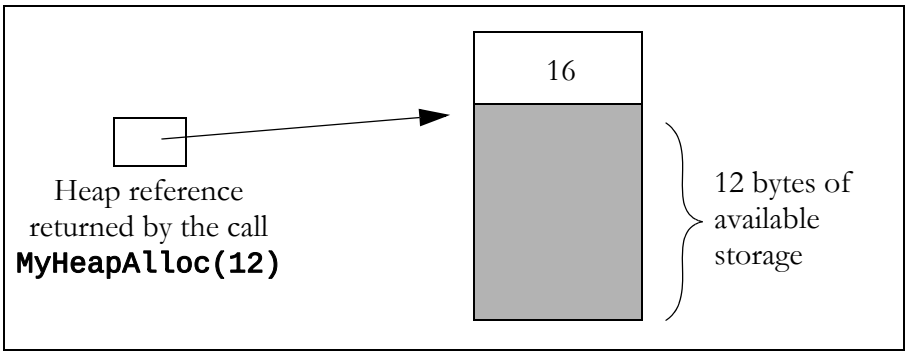

Figure 1: Result of a call to MyAlloc(12)

Unused storage in the JVM heap is organized as a linked-list of free blocks. Each free block begins with 8 header bytes. Four of those bytes are used for the block size, and the other 4 to hold a reference to the next block in the free list. Figure 2 shows a representation of the free list.

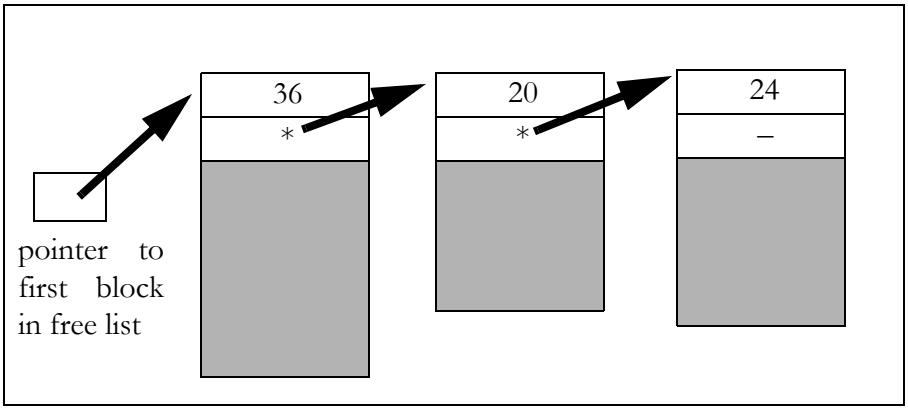

Figure 2: The free list

A call to **`MyHeapAlloc`** causes it to search the free list for the first block which is large enough to satisfy the request. The current implementation uses the first block it finds. If that block is just a little bit larger than needed, the extra bytes are simply wasted. (I.e. the caller receives a block with some extra bytes at the end which are not used.) If the block is a lot too large, the block is split into two. One part is returned to the caller, and the other part is put back into the free list.

The Assignment Requirements

- Your garbage collector should implement a Mark-Sweep strategy. It will follow root pointers to mark all active blocks in the JVM heap. It will then sweep over the heap and return unmarked blocks to the free list.
- If two blocks in the free list are adjacent in memory, they must be re-combined into a single block.
- The root pointers are heap reference values in the JVM stack. Unfortunately, the datatype information for these values is hard to obtain. Therefore, your garbage collector should implement the conservative gc strategy of Boehm and Weiser[1]. This implies that you will perform stringent checks on all values on the JVM stack and, if they pass the checks, assume that they must be pointers. (Occasionally, a value which is not a pointer will pass the checks and it is your responsibility to ensure that such an occurrence will not create problems, other than possibly missing some garbage.)
- It is possible that local C variables in the JVM may contain root pointers. For example, if the call to **`MyHeapAlloc`** in the following C code:

  ```
  x = JVM_PopReference(); /* pop a heap reference */
  y = MyHeapAlloc(12);
  ```

  causes a garbage collection then the C variable **`x`** has to be considered as a root pointer. It is part of your assignment to visually inspect all the calls to **`MyHeapAlloc`** and verify that such situations are not present. If they are, you must correct the code. For example, the two line example above can be corrected by moving the **`MyHeapAlloc`** call to precede the assignment to **`x`**.
- The **`gc`** function is required to update the variables **`gcCount`**, **`totalBytesRecovered`**, and **`totalBlocksRecovered`**. The **`gcCount`** variable is incremented each time a garbage collection occurs. The **`totalBytesRecovered`** variable should hold the total number of bytes returned to the free list by all garbage collections. The total includes the length prefix on each block. The **`totalBlocksRecovered`** variable is simply the count of the blocks discovered to be unused in the course of all garbage collections.

Further Details and Some Help

The Mark phase of a garbage collection starts with *probable* root pointers obtained from the JVM stack (where all local variables of active methods are held). Each such pointer should be verified as much as possible. If it passes the checks, then it references a block of storage in the JVM heap which has a plausible size prefix and which begins with a valid **`kind`** field.

The allocated blocks on the heap are instances of the datatypes defined in the **`jvm.h`** file. We will go through the processing of a class instance to see what needs to be done. If the pointer **`p`** refers to a block used for a class instance, then the storage layout of the block corresponds to the C declaration for **`ClassInstance`** (defined in **`jvm.h`**). The **`kind`** field contains the characters '**`INST`**' and the **`ct`** field contains a pointer to a **`ClassType`** object (there is one per loaded class), which is allocated on the JVM heap and which therefore needs to be marked. Finally, the block contains an array named **`instField`** which has one field for each field in the class instance. The size of this array must be deduced from the size of the block. Some C code which deduces the size is as follows.

---

1. This topic is coming soon in the course. The slides can be found in the set labelled **`GC-part9`**. The original paper is in **`Resources/Reading Material`** under the name **`Boehm-Weiser-1988.pdf`**.

```
ClassInstance *p = ...;  /* reference to the block to be marked */
assert(p->kind == CODE_INST);
int *sizeP = (int*)p - 1;  /* pointer to the size prefix */
int numElements = (*sizeP - ((u1*)&p->instField - (u1*)p));
```

and then the marking phase can continue with

```
for( i = 0;  i < numElements;  i++ )
    Mark(p->instField[i]);
```

Note that some fields will contain heap references and some will not. The JVM does have sufficient datatype information to determine which class fields hold heap references. However it will take a lot of programming to extract that information, so you are not required to do this. Instead, the conservative garbage collection strategy will be sufficient.

There are other kinds of objects in the JVM heap. These are recognized by their different codes in the kind field. These other codes are **'ARRA'** for an array of references to other heap objects, **'ARRS'** for an array on the heap whose elements have simple values (e.g. int values), **'STRG'** for an instance of the **String** type, **'SBLD'** for an instance of the **StringBuilder** type, and **'CLAS'** for an instance of type **Class** (which contains the metadata about a loaded class plus storage for its static fields). Your garbage collector must handle these kinds of objects appropriately.

Note further that your marking phase must handle two kinds of references to blocks on the heap. One kind is declared as type **HeapPointer** and the other kind as type **ClassType***. The former is implemented as an integer offset from the start of the heap (and is guaranteed to fit in 32 bits), while the latter is a memory address and may require 64 bits on some platforms. Conversion between the two kinds of reference is accomplished by the macros **REAL_HEAP_POINTER** and **MAKE_HEAP_REFERENCE** which are defined in **jvm.h**. (If our JVM were to be re-designed, it would probably be an improvement to use the **HeapPointer** type for all heap references.)

Two last notes:

- No mark bit has been provided in the blocks on the heap. This is a detail left to you. A possibility is that you take away one bit from the size prefix attached to each block. For example, you can use the sign bit of the size field as a mark bit. This will limit the size of a block to 2^{31} bytes, but that is unlikely to cause a noticeable problem. (Any way of providing mark bits which makes the blocks in the heap larger is unacceptable.)
- None of the instances of type **Class** can be garbage collected. (In a real JVM, these objects persist as long as the class loader which loaded the class is still live; since our JVM only supports one class loader, we will never be able to collect these objects.)
  The implications are (1) that the pointer **FirstLoadedClass** in file **ClassResolver.c** must be treated as a root variable; (2) every **Class** instance in that chain of loaded classes must be tagged as live; and (3) values in the static fields of class types must be checked to see if they are pointers to objects in the JVM heap.

Integration with Assignment 1

It is your choice as to whether you want to keep adding to your Assignment 1 result or whether you want to re-download all the code (without Assignment 1 enhancements) and treat this as a completely independent assignment. There will be no explicit testing of bytecode verification, but you may want to disable verification in case it interferes with testing your garbage collector.

Testing

It is straightforward to construct a test program which creates garbage. For example, code like

```
for( int i=0; i<100; i++ ) {
    A x = new A();
}
```

will create 100 inaccessible blocks on the heap. More selective creation of garbage can be accomplished with arrays – e.g., fill an array with references to new objects and then assign **`null`** to alternate elements.

The interpreter has a command-line parameter, **`-H`**, which sets the heap size. The default size is 10240 bytes. You can cause the heap to be exhausted more quickly if you make it much smaller by starting up the JVM with a command line like this:

```
MyJVM -W -H1000 Example
```

You can also force a garbage collection in your Java program by allocating a very large array or by making an explicit call to **`System.gc()`**.

Finally, the tracing option **`-Th`** causes calls to **`MyAlloc`** to be traced if you need to see where and how fast the heap storage is getting used up. (You may want to insert statements for generating trace output in your garbage collection code too.)

Rules for performing the work and for submission

- You may form teams of two to three people for performing the work. (Two is preferred.) All team member names must appear in a comment at the beginning of every source code file which you create or modify for Assignment 2. The teams do not need to be the same as in Assignment 1.
- The supplied JVM source code should compile and run on cygwin under a 32-bit Windows platform and on a 64-bit Linux platform. Your code must work correctly on at least the 64-bit Linux platform.
- Submit *all* the source code files as a ingle zipfile. A **`Makefile`** which can be used to build the executable file on Linux is desirable and will be used if provided. If not provided, a command which compiles all files whose names match the **`*.c`** pattern will be used.
- A link for electronic submission will be added to the course webpage.