Due: See website for due date.

What to submit: See website.

The theme of this exercise is automatic memory management, leak detection, and virtual memory.

1. Understanding valgrind's leak checker

Valgrind is a tool that can aid in finding memory leaks in C programs. To that end, it performs an analysis similar to the "mark" phase of a traditional mark-and-sweep garbage collector right before a program exits and identifies still reachable objects and leaks. Note that at this point, the program's main function has already returned, so any local variables defined in it have already gone out of scope.

For leaked (or lost) objects, it uses the definition prevalent for C programs: these are objects that have been allocated but not yet freed, and there is no possible way for a legal program to access them in the future.

Read Section 4.2.8 Memory leak detection in the Valgrind Manual [URL] and then construct a C program `leak.c` that, when run with

```
valgrind --leak-check=full --show-leak-kinds=all ./leak
```

produces the following output:

```
==44047== Memcheck, a memory error detector
==44047== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==44047== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==44047== Command: ./leak
==44047== Parent PID: 43753
==44047==
==44047==
==44047== HEAP SUMMARY:
==44047==     in use at exit: 48 bytes in 6 blocks
==44047==   total heap usage: 6 allocs, 0 frees, 48 bytes allocated
==44047==
==44047== 8 bytes in 1 blocks are still reachable in loss record 1 of 6
==44047==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==44047==    by 0x401138: main (leak.c:11)
==44047==
==44047== 8 bytes in 1 blocks are still reachable in loss record 2 of 6
==44047==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==44047==    by 0x401150: main (leak.c:12)
==44047==
==44047== 8 bytes in 1 blocks are still reachable in loss record 3 of 6
==44047==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==44047==    by 0x401167: main (leak.c:13)
==44047==
==44047== 8 bytes in 1 blocks are indirectly lost in loss record 4 of 6
==44047==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==44047==    by 0x401182: main (leak.c:16)
```

```
==44047==
==44047== 8 bytes in 1 blocks are indirectly lost in loss record 5 of 6
==44047==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==44047==    by 0x40119D: main (leak.c:17)
==44047==
==44047== 24 (8 direct, 16 indirect) bytes in 1 blocks are definitely lost
                                     in loss record 6 of 6
==44047==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==44047==    by 0x401174: main (leak.c:15)
==44047==
==44047== LEAK SUMMARY:
==44047==    definitely lost: 8 bytes in 1 blocks
==44047==    indirectly lost: 16 bytes in 2 blocks
==44047==      possibly lost: 0 bytes in 0 blocks
==44047==    still reachable: 24 bytes in 3 blocks
==44047==         suppressed: 0 bytes in 0 blocks
==44047==
==44047== For lists of detected and suppressed errors, rerun with: -s
==44047== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

(the line numbers in your reconstruction need not match, but the LEAK summary should (including the number of blocks and number of bytes shown.)

2. Reverse Engineering A Memory Leak

In this part of the exercise, you will be given a post-mortem dump of a JVM's heap that was obtained when running a program with a memory leak. The dump was produced at the point in time when the program ran out of memory because its live heap size exceeded the maximum, which can be accomplished as shown in this log:

```
$ java -XX:+HeapDumpOnOutOfMemoryError -Xmx64m OOM
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid2353427.hprof ...
Heap dump file created [89551060 bytes in 0.379 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.base/java.util.Arrays.copyOf(Arrays.java:3720)
        at java.base/java.util.Arrays.copyOf(Arrays.java:3689)
        at java.base/java.util.PriorityQueue.grow(PriorityQueue.java:305)
        at java.base/java.util.PriorityQueue.offer(PriorityQueue.java:344)
        at OOM.main(OOM.java:19)
```

Your task is to examine the heap dump (oom.hprof) and reverse engineer the leaky program.

To that end, you must install the Eclipse Memory Analyzer on your computer. It can be downloaded from this URL. Open the heap dump.

Requirements

- Your program must run out of memory when run as shown above. You should double-check that the created heap dump matches the provided dump, where "matches" is defined as follows.
- The structure of the reachability graph of the subcomponent with the largest retained size should be similar in your heap dump as in the provided heap dump. (Other information such as the content of arrays may differ.)
- You will need to write one or more classes and write code that allocates these objects and creates references between them. You should choose the **same field and class names** in your program as in the heap dump, and no extra ones (we will check this). Think of field names as edge labels in the reachability graph.
- You should investigate which classes from Java's standard library are involved in the leak.

Hints

- The program that was used to create the heap dump is 22 lines long (without comments, and including the main function), though your line numbers may differ.
- Static inner classes are separated with a dollar sign $. For instance, `A$B` is the name of a static inner class called `B` nested in `A`. (Your solution should use the same class names as in the heap dump.)
- Start with the "Leak Suspects" report, then look in Details. Use the "List Objects ... with outgoing references" feature to find a visualization of the objects that were part of the heap when the program ran out of memory.
- The "dominator tree" option can also give you insight into the structure of the object graph. Zoom in on the objects that have the largest "Retained Heap" quantity.
- Use the Java Tutor website to write small test programs and trace how the reachability graph changes over time.
- Do not forget the `-Xmx64m` switch when running your program, or else your program may run for several minutes before running out of memory, even if implemented correctly. (If implemented incorrectly, it will run forever.)
- Do not access the `oom.hprof` file through a remote file system path such as a mapped Google drive or similar. Students in the past have reported runtime errors in Eclipse MAT when trying to do that. Instead, copy it to your local computer's file system first as a binary file. The SHA256 sum of `oom.hprof` is

  `04df06c33e684cc8b0c4e278176ccca885d0abd71fb506e29ad25d8c331a1efa`

3. Using mmap to identify loadable segments in a ELF binary

To practice the use of `mmap`, and also to deepen our understanding of the role of the ELF object file format when loading executables into memory, you will write a short program to find and output the information contained in an ELF executable that instructs the OS's loader which segments to load from the executable and where to place them in the process's virtual address space.

Your executable should be called `loadablesegments` and it should output a recipe for the OS loader. A sample use, which would output `loadablesegments`'s own segments, is shown below:

```
$ ./loadablesegments ./loadablesegments
Loading recipe:
1: load range 0-1728 from the executable to virtual addresses
    0x400000-0x4006c0 and map it read-only
2: load range 4096-5109 from the executable to virtual addresses
    0x401000-0x4013f5 and map it executable
3: load range 8192-8652 from the executable to virtual addresses
    0x402000-0x4021cc and map it read-only
4: load range 11776-12364 from the executable to virtual addresses
    0x403e00-0x40404c and map it read-write
```

(Line breaks were inserted for readability, your program should output one line for each segment; note that the actual ranges may slightly differ for your implementation.)

Your program should use only the `open(2)`, `fstat(2)`, and `mmap(2)` system calls (plus any system calls needed to output the result, such as `write(1)` via `printf`.)

Do not use `read(2)` (or higher-level functions such as `fread(3)`, etc. that call `read()` internally).

The ELF file format is described, among other places, in the Executable and Linkable Format (ELF) Specification. Our rlogin machines have a man page in Section 5: `elf(5)` which documents predefined C structures available in the `<elf.h>` header file you may include. The names used below refer to these structures.

Use the following algorithm:

- Open the file with `open(2)` in read-only mode.
- Use `fstat(2)` to determine the length of the file.
- Use `mmap(2)` to map the entire file into memory in a read-only way.
- You will find the ELF header (`Elf64_Ehdr`) at the beginning of the file.
- You will find an array of program headers (`Elf64_Phdr`) at offset `e_phoff`.

- Iterate over this array, examining each program header. Skip all program headers whose type is not `PT_LOAD`.
- For each program header with type `PT_LOAD`, output one line as shown in the example. Read the man page `elf(5)` to learn the meaning of the `p_flags`, `p_offset`, `p_filesz`, and `p_vaddr` fields.

Simplifying assumptions/hints:

- You may assume a 64-bit little-endian ELF executable, and you may assume that your program is executing on a little-endian machine.
- You may use pointer arithmetic on `void *` pointers, which uses a stride of 1 byte (i.e., it assumes that `sizeof(void)==1`.
- If the given file is not a well-formed ELF executable then the behavior of your program can be undefined.