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 list the entries in a ZIP file

Write a short program `zipdir` that displays the list of entries inside a zip file whose name is passed to the program as its first argument. For each entry it should also print its compression ratio as a percentage rounded to the nearest tenth of a percent. The compression ratio is defined as the ratio of the compressed size to the uncompressed size. A sample use would be:

```
$ ./zipdir heap.zip
heap1.dot                    41.9%
heap1.in                     66.0%
heap1.out                   100.0%
heap1.png                    89.4%
heap2.dot                    36.8%
heap2.in                     59.5%
heap2.out                   100.0%
heap2.png                    92.5%
heap3.dot                    37.0%
heap3.in                     59.2%
heap3.out                   100.0%
heap3.png                    92.0%
```

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 ZIP file format is described, among other places, on the WikiPedia page https://en.wikipedia.org/wiki/ZIP_(file_format)

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.
- scan from the back of the file until you find the beginning marker of the End of Central Directory Record (EOCD).
- extract the number of central directory records in this zip archive and the start offset of the central directory.
- Then, starting from the start offset of the central directory, examine each central directory file header and output the filename contained in it, along with the compression ratio. (Hint: use the following format string for printf:

  ```
  printf ("%-25.*s %5.1f%%\n", namelength, name, ratio);
  ```

- Skip forward to the next central directory record by advancing $46 + m + n + k$ bytes where n is the length of the filename, m is the extra field length, and k is the file comment length contained in each central file directory header.

Simplifying assumptions/hints:

- All multibyte integers in a ZIP file are stored in little-endian order, and — for the purposes of this exercise — you may assume that the host byte order of the machine on which your program runs is little endian as well.
- You may use pointer arithmetic on `void *` pointers, which uses a stride of 1 byte (i.e., it assumes that `sizeof(void)==1`. To access 16-bit or 32-bit values, use `uint16_t *` and `uint32_t *`, respectively under the assumption of little endian host byte order.
- If the given file is not a well-formed ZIP archive then the behavior of your program can be undefined.
- Be sure to handle empty zip files that have no entries.