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.

For leaks, it uses the definition prevalent for C programs: 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:

```
==4162265== Memcheck, a memory error detector
==4162265== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4162265== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==4162265== Command: ./leak
==4162265==
==4162265==
==4162265== HEAP SUMMARY:
==4162265==     in use at exit: 950 bytes in 5 blocks
==4162265==   total heap usage: 5 allocs, 0 frees, 950 bytes allocated
==4162265==
==4162265== 100 bytes in 1 blocks are possibly lost in loss record 1 of 5
==4162265==    at 0x484482F: malloc (vg_replace_malloc.c:431)
==4162265==    by 0x40117C: main (leak.c:20)
==4162265==
==4162265== 200 bytes in 1 blocks are still reachable in loss record 2 of 5
==4162265==    at 0x484482F: malloc (vg_replace_malloc.c:431)
==4162265==    by 0x401137: main (leak.c:12)
==4162265==
==4162265== 200 bytes in 1 blocks are still reachable in loss record 3 of 5
==4162265==    at 0x484482F: malloc (vg_replace_malloc.c:431)
==4162265==    by 0x401148: main (leak.c:13)
==4162265==
==4162265== 300 bytes in 1 blocks are indirectly lost in loss record 4 of 5
==4162265==    at 0x484482F: malloc (vg_replace_malloc.c:431)
==4162265==    by 0x401167: main (leak.c:16)
==4162265==
==4162265== 450 (150 direct, 300 indirect) bytes in 1 blocks
                                  are definitely lost in loss record 5 of 5
```

```
==4162265==    at 0x484482F: malloc (vg_replace_malloc.c:431)
==4162265==    by 0x401159: main (leak.c:15)
==4162265==
==4162265== LEAK SUMMARY:
==4162265==    definitely lost: 150 bytes in 1 blocks
==4162265==    indirectly lost: 300 bytes in 1 blocks
==4162265==      possibly lost: 100 bytes in 1 blocks
==4162265==    still reachable: 400 bytes in 2 blocks
==4162265==         suppressed: 0 bytes in 0 blocks
==4162265==
==4162265== For lists of detected and suppressed errors, rerun with: -s
==4162265== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

(the line numbers 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_pid4144715.hprof ...
Heap dump file created [102988209 bytes in 0.594 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.base/java.util.TreeMap.put(TreeMap.java:575)
        at java.base/java.util.TreeSet.add(TreeSet.java:255)
        at OOM.main(OOM.java:8)
```

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 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 11 lines long (without comments, and including the main function), though your line numbers may differ.
- 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

  `f56ec879711a2a3d55fac29e11650b9cb73409fd558197befbca0ff01470dc7d`

3. Using mmap to list the unresolved symbols in a ELF binary

To practice the use of `mmap`, and also to deepen our understanding of the ELF object file format, you will write a short program to display the list unresolved symbols in a dynamically linked ELF binary or object module (in other words, the symbols that will need to be resolved by the dynamic linker).

Your executable should be called `elfreader` and it should output the list of unresolved symbols for any ELF files passed on the command line. A sample use, which would output `elfreader`'s own unresolved symbols, is shown below:

```
$ ./elfreader ./elfreader
U __libc_start_main@GLIBC_2.34
U puts@GLIBC_2.2.5
U mmap@GLIBC_2.2.5
U printf@GLIBC_2.2.5
U __assert_fail@GLIBC_2.2.5
```

```
U strcmp@GLIBC_2.2.5
U open@GLIBC_2.2.5
U fstat@GLIBC_2.33
```

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 section headers (`Elf64_Shdr`) at offset `e_shoff`.
- Identify the section header with index `e_shstrndx` and retrieve its section header offset `sh_offset`. This will provide an offset that represents the beginning of the section name string table. We will need this string table to find the actual string table with the strings describing the module's symbol names.
- Iterate through all section headers to and find a section with `sh_type` being equal to `SHT_SYMTAB`. This is the symbol table.
- If no such entry was found, output

  ```
  No symbol table found.  Stripped executable?
  ```

  and stop.
- Also while iterating through the section header table, identify a section with type `SHT_STRTAB` and where the `sh_name` field points to a string `".strtab"`. Note that the `sh_name` field contains a numeric offset that must be added to the beginning of the section name string table found above before applying `strcmp`. Retrieve the `sh_offset` field of this (second) string table for later use.
- Finally, identify the number of symbols in the symbol table by dividing the size of the symbol table section (`sh_size`) by the size of a single instance of `Elf64_Sym`.
- Iterate through each symbol table entry, starting in memory at the `sh_offset` associated with the symbol table, treating each entry as a `Elf64_Sym` object. For all entries such that `st_shndx` is equal to `STN_UNDEF` check if the condition

`(ELF64_ST_BIND(st_info) == STB_GLOBAL)` holds, if so, output a line with the letter `U`, followed by a space, followed by the name of the symbol. `st_name` is an offset in the (second) string table identified above that points to a zero-terminated string that can be passed to `printf()`.

Simplifying assumptions/hints:

- You may assume a 64-bit little-endian ELF executable or object file, 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 ZIP archive then the behavior of your program can be undefined.