Due: Friday, April 4, 2014. 11:59pm (no extensions).

What to submit: Upload a single ASCII file with your answers.

Part 1 of this exercise should be done on the rlogin machines, using the 64-bit compiler. Part 2 of this exercise requires the Eclipse Memory Analyzer tool (MAT). This tool is installed on the lab machines in `/home/courses/cs3214/software/mat/-MemoryAnalyzer`, but you may prefer to install it on your own machine for better performance. It can be downloaded from http://www.eclipse.org/mat/downloads.php.

You should use valgrind from the directory below since the default version installed on rlogin can be buggy:

`/home/courses/cs3214/valgrind-3.7.0-install/bin/valgrind`

Memory management bugs are a common problem when using explicit or automatic storage allocators. Familiarity with available diagnosis tools and knowing their capabilities and limitations are crucial skills for any productive programmer.

1. Understanding valgrind

Memory analysis tools for type-unsafe languages such as C are now in wide use, helping programmers pinpoint many previously hard-to-find bugs. Examples include the open-source program valgrind and commercial products such as Purify. These tools use binary instrumentation to intercept a program's accesses to memory (though not its use of registers). They maintain a data structure called *shadow memory* that records for each bit in the actual program's memory whether it is part of the address ranges a program is legally allowed to access, and whether it has been written to before being read or used in a conditional. In addition, they replace the memory allocator with their own implementation in order to intercept when memory regions are deallocated and subsequently rededicated to newly allocated memory blocks.

The directory ~cs3214/public_html/spring2014/exercises/memory1/bugs contains 14 files `bugs1.c` to `bugs11.c` and `bugs14.c` to `bugs16.c`. Each of these files contains one or more memory related bugs. A script `compileall.sh` compiles each file. Each file is compiled once without any `-O` switch, and once with the `-O2` switch. In either case, `-Wall` switch is also enabled. You should examine each file, identify whether the error contained in the file was detected by the compiler and/or by valgrind. If the error was detected by the compiler, provide the warning or error printed. If the error was detected by valgrind, copy and paste the relevant error message here. If it was not detected by valgrind, explain why not.

For example, for `bugs1`, you would provide the following answers: (Note that `-Wall` switch is also enabled.)

```
bugs1:  Use of uninitialized local variable
bugs1/-O0:
```

```
Detected by compiler:  Yes
"bugs1.c:11: warning: 'uninitialized' is used uninitialized
 in this function"
Detected by valgrind:  Yes,
"Use of uninitialised value of size 8"

bugs1/-O2:
Detected by compiler:  Yes
"bugs1.c:11: warning: 'uninitialized' is used uninitialized
 in this function"
Detected by valgrind:  No:
Reason: variable was allocated in a register, so no memory
 access to uninitialized memory for valgrind to see.
```

2. Tracking Down Out Of Memory Conditions

In type-safe languages such as Java or $C^{\#}$, many sources of memory-related errors have been eliminated. Memory leaks remain as a serious source of error you will encounter in your practice as a programmer. Memory leaks increase garbage collection frequency because they reduce the amount of free heap space, and they make every full garbage collection more expensive since leaked objects will typically be tenured in the oldest generation. Eventually, they lead to an out-of-memory error if the size of the uncollectable live heap exceeds the virtual machine's maximum heap size.

Tracking down memory leaks is often difficult in today's framework-based applications that consist of multiple layers provided by different components or libraries. When an out-of-memory error occurs, the programmer must determine why the size of the live heap exceeded the JVM's limit. Possible reasons include:

- The (legitimate) size of all data the application attempted to operate on was larger than the maximum live heap size the JVM allows.
- The application erroneously attempted to allocate (and keep alive!) objects without regard to their memory consumption. An example may be an infinite loop that grows a buffer in each iteration.
- The application failed to zero out references to objects or structures that it will no longer access, preventing the collector from freeing them.

Each of these reasons demands a different strategy: if the problem size is too large, the maximum amount of memory allocated to the JVM must be increased (e.g., by using `-Xmx` or by choosing a larger platform), or the problem size must be reduced. If allocation is excessive, it must be throttled or avoided, if the application logic allows such throttling. (In some cases, excessive allocation results from the application not handling error conditions correctly.)

If leaked objects are the cause, the responsible code module must be identified, using such clues as the class name(s) and chain of references to the object.

In this exercise, you are asked to use the Eclipse Memory Analyzer to investigate a post-mortem dump heap obtained from a real-life J2EE server during development. The application the developer was developing is a multi-tiered application that communicates with a separate XML database via a socket connection. It also uses the ZK web application framework (zkoss.org). The J2EE server is Tomcat (tomcat.apache.org).

In the given scenario, the JVM suddenly ran out of memory. When that happened, users would be greeted with a page displaying an error message "PermGen space out of memory error." Using the JVM's `-XX:+HeapDumpOnOutOfMemoryError` flag, the developer was able to obtain a dump of the heap at that point in time.

Your task is to examine the heap dump and find out as much as you can about the cause of the out-of-memory condition. You should use your knowledge of JDK classes and their implementation if appropriate. A copy of the application's source code is included.

Based on your analysis, recommend a strategy on how to deal with the problem.

The heap dump can be found in the directory ~cs3214/public_html/spring2014/exercises/-memory1/leak3. A snapshot of the source code of the application is included. The directory ~cs3214/public_html/spring2014/exercises/memory1/leak1 contains an example to help you interpret the information displayed by MAT for a known, small program.