TotalView Debugger Training Lab Manual Answers

Lab 1: Debugger Basics Startup, Basic Process Control & Navigation

- Question 1: Where can you find the state of your currently focused process?
    - At the top of the Process Window below the toolbar and above the stack trace and Stack Frame. Info about all the threads that make up the process is available in the Threads Tab.
- Question 2: What did choosing the Step Instruction command do, if anything?
    - The process advanced by one machine instruction.
- Question 3: Why doesn't it look like anything changed?
    - The target program is still executing "in the same source code line" as it was before the step.
- Question 4: What can you do to see the effect of stepping an instruction?
    - You can look at the assembly in the source code window by Viewing -> Source as -> Assembly or Both.
- Question 5: Why didn't you step into the **printf()?**
    - Because it is a system library function that is not compiled for debugging.
- Question 6: Where did the output from the **printf()** call go?
    - On the console where we launched TotalView Debugger.
- Question 7: What did this do?
    - The target program has just returned from the function call.
- Question 8: Why did TVD display this message?
    - Because the **scanf()** on the line in question doesn't complete until there is some input.
- Question 9: What will happen if you enter input on **stdin**?
    - The **scanf()** will return and the message will go away.
- Question 10: What will happen if you press cancel?
    - The target program will be halted in the middle of trying to read data. We will probably see assembly since we will be looking at library functions.

Lab 2: Viewing, Examining, Watching and Editing Data

- Question 1: What kind of information is displayed within the Stack Frame Pane?
    - Function parameters, local variables, and registers. Scalar values are listed directly, aggregate types are just listed with the type info.
- Question 2: What does the bold text mean?
    - It shows values that are editable.
- Question 3: What does the block designation mean?
    - Blocks refer to scopes. Variables such as **i** and **j** are only valid within the **for** loops where they are defined.
- Question 4: Why does the type field indicate "**struct Cylinder**" rather than "**class Cylinder**"?
    - Because classes are a special type of structures and they are represented that way in the debug information.
- Question 5: When would you want to use the expression list as opposed to a Variable Window or Tooltip?
    - When you want to monitor several expressions at one time, possibly coming from different scopes.
- Question 6: Why does the **last value** field report the value of j as 1 instead of 19?
    - Because the last value is the last value displayed by the debugger - not the last value that the program had for the variable.
- Question 7: What does **Stale** mean and why are the Expression List and Variable Windows reporting this?
    - It means that the memory locations are no longer 'active' or in scope.
- Question 8: Why doesn't TVD display this as an array?
    - The debug information only tells the debugger that the variable is a pointer. The debugger has no idea if it points to one, two, or many **compound_t structs**.
- Question 9: What does **Sparse** mean in the **Address** field?
    - The elements that are shown are not adjacent in memory.
- Question 10: When will a watchpoint trigger?
    - When the memory location identified is written to and the value changes. (Writing zero over a location that already has zero will not trigger a watchpoint on that location.)
- Question 11: What precautions do you need to take when planting a watchpoint on a local variable?
    - Be aware that when the function returns, the memory associated with the variable will go out of scope. The watchpoint will not automatically be removed at the end of scope.

Lab 3: Examining and Controlling a Parallel Application

- Question 1: Why are all the ranks stopped at exactly the same point in the program? Is this a coincidence?
    - No it is not a coincidence. We've set a breakpoint and run all the processes in the program to this breakpoint.
- Question 2: Note that only a subset of the processes are halted at breakpoint one. Why is this?
    - We've asked the program to stop the entire parallel job when any one of the processes reaches the breakpoint location.
- Question 3: What do the edges on the graph represent?
    - Each edge represents the fact that a function has been called. The call-stack within each process records the set of functions called and the function from which each was called. Each edge has a set of numbers indicating the set of processes that have the specified call event on their call stack.
- Question 4: What do the nodes in the graph represent?
    - Each node represents a function, subroutine, or block that is found on the call stack of one or more of the selected processes. Each function is represented only one time. Arrows to the function indicate times that the function was called. Arrows from the function indicate functions called from within it.
- Question 5: What do you expect to happen if you now select the **call_graph** group in the focus control and hit the **go** button?
    - The set of processes which have the specified call event on their call stack run, those that do not have the event on their call stack do not run.
- Question 6: What do you expect to happen if you change the focus to control group and say **go**?
    - All the processes in the MPI job will run, except the one that is held.
- Question 7: What is a common cause of a deadlock in an MPI application?
    - Mismatched message passing operations.
- Question 8: What features does TotalView provide to help you with this type of problem?
    - The Call Graph, the MPI Message Queue Display, the MPI Message Queue Graph, and cycle detection within the MPI Message Queue Graph.
- Question 9: What does the Message Queue Graph tell you?
    - It graphically represents the contents of the MPI message queues: Pending Sends, Unexpected Messages, and Posted Receives.
- Question 10: Does the Message Queue Graph show you all messages ever sent or just the pending ones?

    - It only shows messages that are currently pending. As a message is received it is removed from the queue.
- Question 11: Why does it focus you on assembler code?
    - Because you are looking at the MPI starter process -- which was most likely not built with debugging information (symbol tables and source code line number info).

Lab 4: Exploring Heap Memory in an MPI Application

- Question 1: In what circumstances would you have to link with the memory debugging library (**libtvheap.so**)?
    - If you are attaching to a process (or set of processes) after they have already started running. This is often the case with applications that are started from a script.
- Question 2: Where can you find the same information as is displayed in the Memory Event Details Window's Event Location Tab?
    - The call stack panel on the Process Window.
- Question 3: Where can you find the same information as displayed in the memory event details window's Allocation, Deallocation, and Block details tabs?
    - The block properties dialog for the block that is involved in the event.
- Question 4: What are the stack frames above the frame for **free()** and why does TotalView report that the process is at a breakpoint? Is this a breakpoint you can disable or delete?
    - These are the internal pieces of the HIA library. Functions on which "Magic" breakpoints can be placed are used to communicate between the HIA and the debugger. No, you cannot enable or disable magic breakpoints.
- Question 5: Since TotalView does not automatically focus you to a process when it receives a memory event what can you do to make sure that you do not miss an event? How can you recover event information should you miss it?
    - Check the Manage Processes > Process Events Report in the MemoryScape Window.
- Question 6: Why does the process list in the bottom left corner only have one process?
    - Because the selected process hasn't generated any memory events.
- Question 7: What kind of memory events and errors can MemoryScape provide events for?
    - Ones for which there is a free() (or delete or deallocate) operation. For example, double free, freeing something on the stack, etc.
- Question 8: What types of memory corruption can the Heap Interposition technology help you with?
    - Reading and writing beyond the bounds of a block allocated in the heap.
- Question 9: When can you get notified that your application has corrupted memory?
    - If you are using Guard Blocks, you are notified when the block is freed. If you are using Red Zones, you are notified immediately when the read or write occurs.

- Question 10: How can you change the bit pattern that TotalView uses to paint the guard regions?
    - Through the Memory Debugging Options Tab, Advanced Options in the MemoryScape window.
- Question 11: Why didn't MemoryScape halt your program at the same time it overwrote the bounds of the array?
    - MemoryScape does not check the validity of every read and write that the program does (and that is a good thing -- the program would run **very** slowly if it did).
- Question 12: Why can't you toggle the enable memory debugging option while the program is running?
    - Because catching memory errors relies on having a complete view of all the heap allocations. If you took a program that was already running and started tracking heap operations half way though you would have incomplete information which would break things like leak detection.
- Question 13: Was the memory beyond the array bounds actually overwritten?
    - No, the Red Zone placed at the boundary denies access to that memory.
- Question 14: Why is there so much empty space with Red Zones?
    - Each allocation must be placed in its own page. We need an extra page in order to detect bounds errors. Thus, to detect bounds errors we need two pages for each allocation.
- Question 15: Why is there so much additional overhead for Red Zones?
    - Red Zones use the operating system's services to manipulate the access permissions of regions of the process's address space. These primitives operate in units of pages.
- Question 16: Why can't a program be continued after a Red Zone event?
    - Since the program attempted a memory access but was prevented from completing it, the state from which the program should continue is undefined.
- Question 17: Why wasn't a Red Zone event triggered as before in the corrupt_data_rz function?
    - That routine uses allocations whose size is outside the Red Zone size range restrictions.
- Question 18: What is different about this latest Red Zones event? Could the memory error in this case have been detected with Guard Blocks?
    - Here, the programming error involves a read from memory, rather than a write to memory as in earlier examples. Guard Blocks cannot detect a read-only overrun.
- Question 19: What are some ways of limiting memory space overhead when using Red Zones?
    - Red Zones can be disabled during a program's startup, and only enabled at a point where the program's memory behavior is suspect. Red Zones

size range restrictions can be used to exclude allocations that are not suspect.

- Question 20: How does MemoryScape define a related heap block?
  - The set of blocks with the same call sequence backtrace at allocation (or deallocation).
- Question 21: Why is it useful to know about related heap blocks?
  - It can help you understand allocation patterns within your program. If all the leaks in a program are related (in this manner) then that gives you a really good clue where to look for the actual bug for which the memory leak is a symptom.
- Question 22: Why doesn't the graph display memory leaks by default?
  - Because leak detection takes an amount of time that varies depending on the memory footprint of the application. Looking at the heap status (without leak detection) is much faster. MemoryScape leaves the choice to the user.
- Question 23: How does the source view organize the information?
  - By process, source code file, function, line number and finally individual blocks.
- Question 24: Do you see anything peculiar with the number of byes and allocations in the processes?
  - One is much larger than the others.
- Question 25: Do the memory blocks allocated in **myMalloc()** all have the same backtrace? Why or why not?
  - No because **myMalloc()** is called in many places in the program.
- Question 26: How does MemoryScape define the allocation focus point for a memory block?
  - It defaults to the "leaf-most" function for which there is debug information.
- Question 27: Under what circumstances is MemoryScape choice of the allocation focus point not optimal?
  - If you wrap **malloc()** with something like **myMalloc()** you might find the locations where **myMalloc()** is called to be much more interesting than the fact that a **malloc()** is called a lot of times from **myMalloc().**
- Question 28: How does MemoryScape define a memory leak?
  - A block which has been allocated, not yet freed, but for which there is nothing in any register or reachable memory which looks like it could be a pointer to the block (or optionally into the block).
- Question 29: How is generating a leak report in the Leak Detection Tab different from detecting leaks in the Heap Graphical or Heap Source Reports?
  - The leak report only shows the leaks (all the statistics will be numbers and bytes of leaked memory).
- Question 30: What rank is clearly using more heap than the others?
  - Rank 1

- Question 31: Does memory debugging have to be enabled in order to view the memory usage reports?
    - No.
- Question 32: Why do the two reports show different values?
    - Because one is including memory that is reported by the kernel as being part of the process, the other is just summing up the allocated blocks. The kernel will generally report more than the sum of the allocated blocks. The difference has to do with how memory is divided into pages, some buffering, and other overhead.
- Question 33: What might the Memory Usage Report be useful for during a debugging session?
    - It gives you an idea how much space is being used for the stack, data, and other sections of memory besides the heap. It also gives a nice overview of memory usage if you are looking at lots of processes at the same time.
- Question 34: Observe the bytes and counts reported for Session 1. Why are they zero?
    - Because this difference view is removing everything that is “the same” between the two. In this program there is a common base of allocations (which you don’t see in the comparison view), and the process with rank 1 does some additional allocations (which you do see).
- Question 35: What allocations are not displayed in the Memory Comparison Report?
    - Those that are in common between the two selected data sources.
- Question 36: Why is there no difference?
    - Because you are comparing the heap information about a live process with a stored copy of that same heap information.
- Question 37: Under what circumstances might you want to compare to the differences between a live process and a postmortem process?
    - You might make a change to the code (or patch in a fix, as we just did) and then want to see what effect it has on the behavior of the program.
- Question 38: What other kinds of memory reports can you generate on a memory debug file?
    - Heap Status (various forms) and the Leak Detection reports.

Lab 5: Batch Mode Debugging with TVScript

- Question 1: What are some ways in which examining these program variables with TVScript is more convenient than the conventional batch debugging practice of inserting print statements into the program?
    - TVScript doesn't require rebuilding the program. TVScript does appropriate formatting automatically, even for rich data types such as the struct in this example.
- Question 2: What stands out in the output of the current estimate of pi (almost_pi) compared to the reference value (ref)?
    - The output of the estimated value has the expected precision, but the reference has far less precision. This leads to the cause of the program's problem: the reference was declared as a float instead of a double.

Lab 6: Reverse Debugging with ReplayEngine

- Question 1: Is debugging still in replay mode? How can you tell?
    - Even though the Go changed the execution direction to forward, the breakpoint was reached at a point within the recorded history, so debugging is still in replay mode. This is indicated by orange highlighting of the source line where historical execution stopped.
- Question 2: What forward debugging actions are analogous to the reverse debugging actions that were used so far?
    - Prev is analogous to Next; UnStep to Step; Caller to Out; and BackTo to RunTo.
- Question 3: What is the most likely reason for the loss of program position information?
    - The program's stack has been overwritten, corrupting the stack trace.
- Question 4: A deterministic bug is possible to find with forward debugging. In this case, once forward debugging had established that the symptom occurs in funcB, the rest of the effort would have been similar to using reverse debugging. (That is, most likely a watchpoint would have been set on arraylength to lead to the root problem.) What are some advantages of reverse debugging in this case?
    - (1) There are numerous executions of funcB with no symptoms (due to recursion in this case, but looping or parallelism could have similar effect). Forward debugging would probably entail examining every execution of funcB, and somehow keeping track of which were benign.
    - (2) With forward debugging, the program would have to be restarted every time the segmentation violation was encountered, plus at least one additional time with a watchpoint set.
- Question 5: Can you characterize how the program is failing to operate as designed? Is the misbehavior repeatable from run to run?
    - One (or more) ranks are failing to sort their subdomain of data. The set of failing ranks changes from run to run. Sometimes there are no failing ranks at all.
- Question 6: What does the warning mean?
    - In this rank, the call to qsort was never executed. (This is because of the conditional at line 142, which is an intentionally inserted bug.)
- Question 7: How does the use of ReplayEngine enhance debugging of non-repeatable bugs?
    - With forward debugging, typically it is necessary to make several runs of the program, each time observing symptoms of the bug in order to guide debugging strategy. If the program fails in different ways from run to run, or only fails rarely, this process can require a good deal of time and luck. With reverse debugging, only one failing run is necessary. All of the program state leading to the bug's symptoms is preserved for analysis, even in a parallel program.