CS341: Computer Architecture Lab

Lab #0: Debugging
Report

Aakriti (190050002)

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
2021-2022

Contents

0.1 Question 1

Aim: Use *gdb* for debugging and develop Pretty Printers for it.

0.1.1 (a) Make Pretty Printer

Here is how the **vheap* is printed by my `prettyprinters.py` -

```
index2HeapIdx is std::unordered_map with 9 elements =
[3] = 2, [2] = 0, [7] = 6, [4] = 3, [6] = 5, [5] = 4, [1] = 7, [8] = 1, [0] = 8
heapIdx2index is std::unordered_map with 9 elements =
[8] = 0, [7] = 1, [6] = 7, [1] = 8, [0] = 2, [2] = 3, [3] = 4, [4] = 5, [5] = 6
Element(s) of heap (starting from 0 indexing) is
heap[0] is 12
heap[1] is 15
heap[2] is 25
heap[3] is 21
heap[4] is 11
heap[5] is 9
heap[6] is 8
heap[7] is 4
heap[8] is 0
capacity is 9
size is 4
```

0.1.2 (b) Detect Bugs in Djikstra

1. First error is in `heap.cpp`, in `VertexHeap::swap` function at line no. 114.
   Original Code: `heapIdx2index[heapidx2] = temp;`
   Modified Code: `heapIdx2index[heapidx1] = temp;`

2. Second error is in `heap.hpp`, while defining left and right children in a heap with 0 indexing.
   Original Code:
   `#define HEAP_LEFT(i) (2*i)`
   `#define HEAP_RIGHT(i) (2*i+1)`
   Modified Code:
   `#define HEAP_LEFT(i) (2*i+1)`
   `#define HEAP_RIGHT(i) (2*i+2)`

0.2 Question 2

Aim: Use `valgrind` for detecting memory leaks and resolve it.

0.2.1 (a) Detect Memory Leaks

There are 6 places where memory leaks occur in this code. They are described as follows -

1. The following is snippet from output shown by valgrind -
   ```
   ==240== 3 bytes in 1 blocks are still reachable in loss record 1 of 6
   ==240== at 0x4C2FB6B: malloc (vg_replace_malloc.c:299)
   ==240== by 0x4ED9A29:  strdup (strdup.c:42)
   ==240== by 0x108909:  addArg (argparse.c:21)
   ==240== by 0x10880E: main (main.c:8)
   ```

   As we can see here, 3 bytes of heap memory allocated at line 21 of `argparse.c` and the memory is never freed. this happens once every time an argument is added and `addArg()` is called in the main. Since, it is called 5 times, $5 \times 3 = 15$ bytes of data is allocated but never freed so 15 bytes of memory leak occurs.

   The line in argeparse which causes 15 bytes of memory leak -
   `argParser.argList[argParser.len++].name = strdup(name);`

2. The following is another snippet from valgrind output -
   ```
   ==532== 96 bytes in 1 blocks are still reachable in loss record 6 of 6
   ==532== at 0x4C2FB6B: malloc (vg_replace_malloc.c:299)
   ==532== by 0x10895D: addArg (argparse.c:27)
   ==532== by 0x108832:  main (main.c:10)
   ```

   Here, 96 bytes of memory is allocated by malloc in line 27 of `argparse.c` for an array of `struct arg` and is never freed. So, another 96 bytes of memory leak occurs.

   In total, $96 + 15 = 111$ bytes. All the leaked memory has been accounted for.

0.2.2 (b) Free Memory

Relevant changes have been made in `argparse.c`, `argparse.h` and `main.c` to free all the allocated heap memory. A new function `void freeMem();` has been created for this due to which, the header file and main file also had to be edited.