Operating Systems

Lab 01 – Team Tchotchkes

Report

Members:

Andrew Montgomery, Deepti Rao, Rebekah Lee

Table of Contents

00: AN INTRODUCTION

Why do we need sorting?

Some real-world applications of sorting include, but are not limited to, sorting of television channels, adding pictures to databases, and indexing the internet's pages for search engines. The ability to sort different data types efficiently is paramount to solving some of the world's biggest data and computing problems.

Some Different Types of Sorting

General forms of sorting include internal and external sorting, in-place and out-of-place sorting, and stable and unstable sorting. Internal sorting is when the data to be sorted is confined enough that it can be placed directly in main memory at a given time, and the sorting process can take place there. This can be very advantageous because the storage disk is not needed. External sorting is basically the opposite. When the data to be sorted is too large to keep in main memory, it is brought into the storage disk or medium. In-place sorting means that the storage location used for the input data is used for the output data at the time of sorting, this gives it a very nice space complexity of O(n).

Sorting stability has to do with whether all the elements are unique within the given list or data structure. A particular algorithm is said to be stable if the relative order of the duplicate keys is maintained through to the sorted output. That is, if keys are equal, then their relative order in the sorted output is the same as that in the input. If a sort is unstable, the keys that are of equal value may occur in any order within the sorted output.

<u>01: ASSUMPTIONS, INTERPRETATION OF SPECIFICATIONS</u>

Some Assumptions:

What were our assumptions for this specification? We start off by assuming that the array or linked lists we are going to be sorting are of relatively small size. Also, they consist only of integers and not floating-point values or any other data type. We will be using 0 as the terminating symbol for our algorithm(s). There will be only non-negative values in our lists to be sorted. We also will be going off the basis that the worst time complexity of Bubble Sort is O(n^2), while the space complexity is in O(1) or constant space, i.e. we will not be creating new values while running through our algorithms.

02: THE OBJECTIVE: WHAT ARE WE TRYING TO DO?

The objective here is to implement bubble sort on nearly-sorted arrays, where performance is good. Also, on arrays and lists of relatively small size. At a high-level, we are trying to familiarize ourselves with the different types of sorting algorithms out there, but on an Assembly language level. The end goal is to attack a problem that we have seen several times before but have to approach it in a new manner. This can be valuable in helping us further our understanding of how computers and programming languages operate in general. We hope to gain new knowledge, understanding and skills through these exercises.

03: INPUT/OUTPUT

The input to our sorting algorithm is 4 integers in an unsorted list. The output will be the same integers, but in sorted order, shown with GDB. We are using two sets of variables to accomplish the linked list implementation. This was the best way that we found to accomplish the functionality of keeping the values in the same address while changing the pointer to each one during the sorting process.

04: RELATING THE LAB TO LECTURES

Through the lectures and class periods, we gained invaluable knowledge and resources for this assignment. Having our lecture notes and slides already saved valuable time in figuring things out. The assembly programming and tracing assignments allowed us to know how to debug and trace our code to accomplish the task at hand. The knowledge provided during class regarding memory and the stack and heap were very helpful as well.

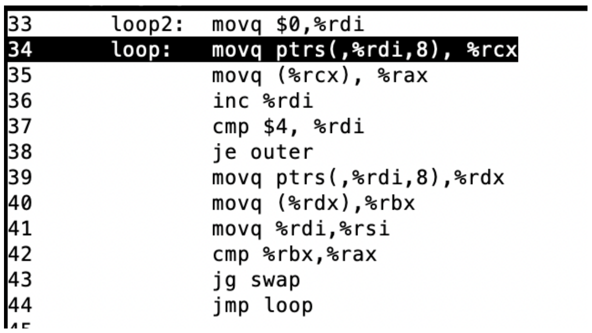
05: STEPS OF OUR ALGORITHM

The algorithm we used follows very closely the bubble sort algorithm from Geeks for Geeks webpage here: https://www.geeksforgeeks.org/bubble-sort/ .

Using nested loops to traverse the loop front to back from each index and swap the values if they are greater than the higher index, and then start the loop again after each swap. This will work because it ensures that we have traversed through the list enough times to cover all the indices and any chance of items being missed.

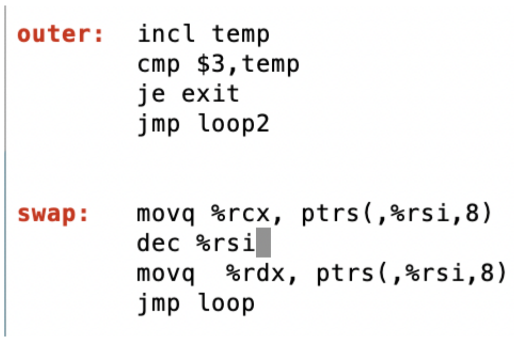
06: MAPPING THE ALGORITHM INTO THE CODE

To map this algorithm into our code, we will start by using rax and rbx registers as placeholders for the adjacent values being compared. Then we will use rdi and rdx for the loop counters. Using rbp for the base address.



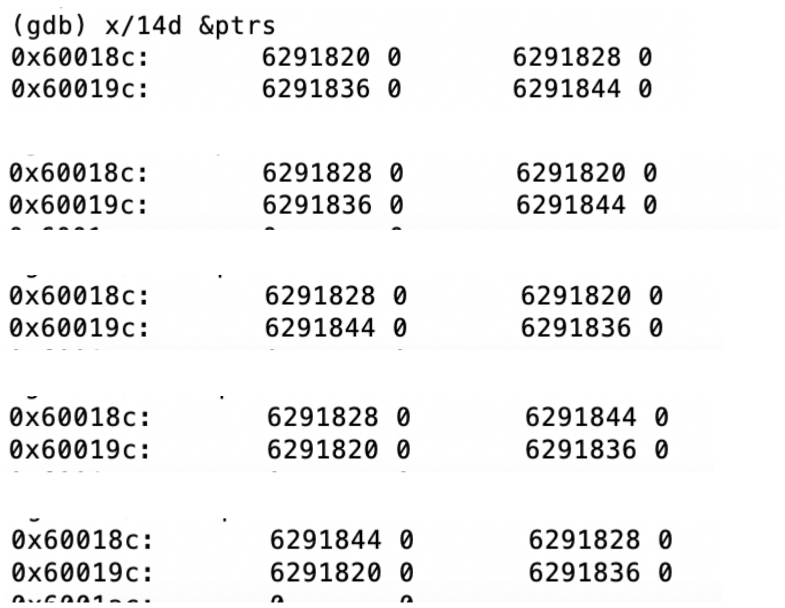
```
loop2:  movq $0,%rdi
loop:   movq ptrs(,%rdi,8), %rcx
        movq (%rcx), %rax
        inc %rdi
        cmp $4, %rdi
        je outer
        movq ptrs(,%rdi,8),%rdx
        movq (%rdx),%rbx
        movq %rdi,%rsi
        cmp %rbx,%rax
        jg swap
        jmp loop
```

The addressing modes to be used in implementing our algorithm are: register addressing, base-pointer addressing, and indirect addressing. Below you will find a snippet of the code form our program to show the outer loop and the swap function.



```
outer:  incl temp
        cmp $3,temp
        je exit
        jmp loop2


swap:   movq %rcx, ptrs(,%rsi,8)
        dec %rsi
        movq  %rdx, ptrs(,%rsi,8)
        jmp loop
```

Our data will be organized into four segments in memory, as we are using four elements in our list.



```
(gdb) x/14d &ptrs
0x60018c:       6291820 0       6291828 0
0x60019c:       6291836 0       6291844 0

0x60018c:       6291828 0       6291820 0
0x60019c:       6291836 0       6291844 0

0x60018c:       6291828 0       6291820 0
0x60019c:       6291844 0       6291836 0

0x60018c:       6291828 0       6291844 0
0x60019c:       6291820 0       6291836 0

0x60018c:       6291844 0       6291828 0
0x60019c:       6291820 0       6291836 0
```

The control flow will be as follows: the pointer section deals with creating our pointers. The data section creates/instantiates our values (integers) and pointer variables. Outer Loop and Inner Loop are the nested loops of our algorithm. The compare section handles swapping of value pointers if they are not in order.

<u>07: TESTING FOR CORRECTNESS</u>

To test for correctness, we used several methods. These include using different types of input, like using arrays with different values, in descending order, etc. Also, when testing that the outer and inner loops are working properly, we would place two items on either side of a 3rd item that need to be swapped. Say for instance, 7 then 9 and then 6. The 6 needs to be swapped before the 7, but the 7 doesn't need to be after the 9, so we need to make sure our algorithm is working in this type of case. These may not cover every single corner case for our program, but we believe they are sufficient for the purpose of this assignment and give us confidence that our functionality is correct.

08: HOW DO WE BUILD AND RUN

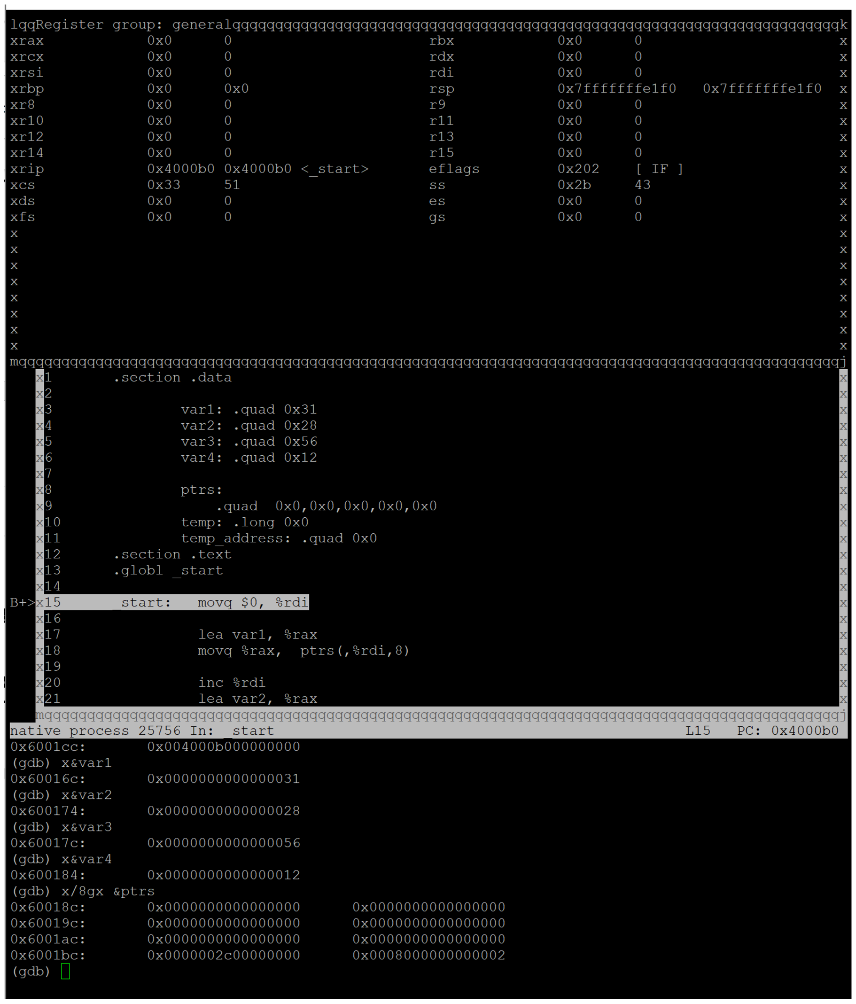
To build our program, we use the same commands from the course of our lectures and assignments in class:

as -g (filename).s -o (filename).o	(assemble with debug flags so we can use gdb)
ld (filename).o -o (filename).x	(link into executable)
gdb -tui	(open gdb)
file ./(filename).x	(open file in gdb)
break _start	(set breakpoint)
layout regs	(show register view)
run	(run program)
step	(step through program to check it is working)
x/#gx &data_section_name	(to show how our values are changing/moving)

This is how we build, link, and run our program to see what it is doing.

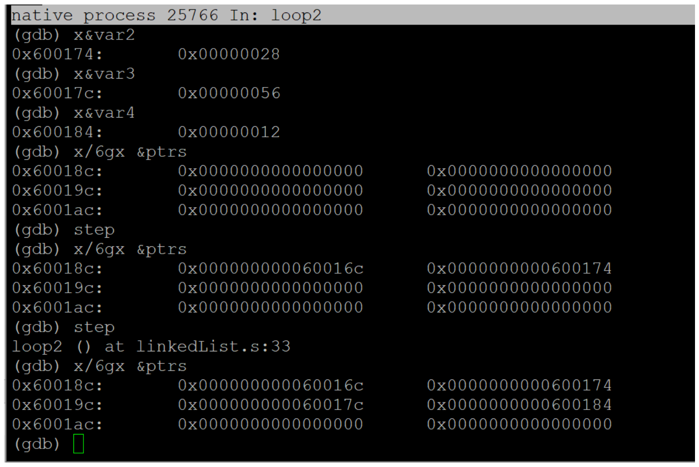
09: DOES IT DO WHAT WE SAID?

GDB register layout and data elements before operation:



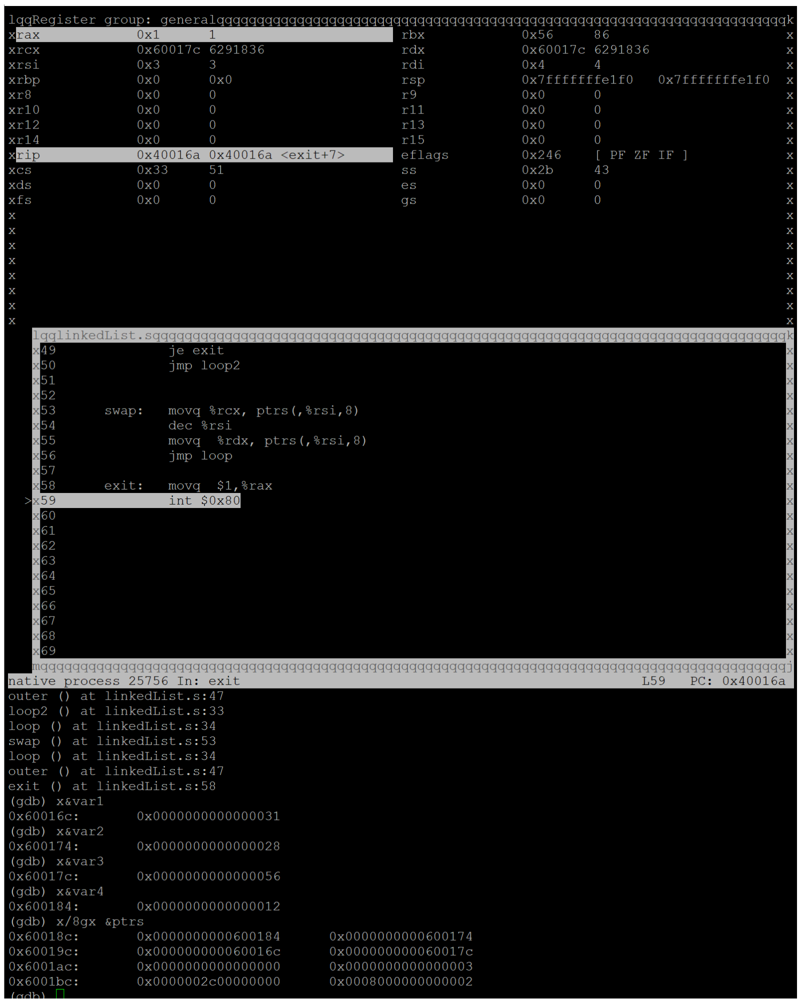
```
lqqRegister group: generalqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
xrax           0x0      0                     rbx            0x0      0                x
xrcx           0x0      0                     rdx            0x0      0                x
xrsi           0x0      0                     rdi            0x0      0                x
xrbp           0x0      0x0                   rsp            0x7fffffffe1f0   0x7fffffffe1f0  x
xr8            0x0      0                     r9             0x0      0                x
xr10           0x0      0                     r11            0x0      0                x
xr12           0x0      0                     r13            0x0      0                x
xr14           0x0      0                     r15            0x0      0                x
xrip           0x4000b0 0x4000b0 <_start>     eflags         0x202    [ IF ]           x
xcs            0x33     51                    ss             0x2b     43               x
xds            0x0      0                     es             0x0      0                x
xfs            0x0      0                     gs             0x0      0                x

        .section .data

                var1: .quad 0x31
                var2: .quad 0x28
                var3: .quad 0x56
                var4: .quad 0x12

                ptrs:
                    .quad  0x0,0x0,0x0,0x0,0x0
                temp: .long 0x0
                temp_address: .quad 0x0
        .section .text
        .globl _start

B+>     _start:   movq $0, %rdi

                  lea var1, %rax
                  movq %rax,  ptrs(,%rdi,8)

                  inc %rdi
                  lea var2, %rax

native process 25756 In: _start                                        L15   PC: 0x4000b0
0x6001cc:       0x004000b000000000
(gdb) x&var1
0x60016c:       0x0000000000000031
(gdb) x&var2
0x600174:       0x0000000000000028
(gdb) x&var3
0x60017c:       0x0000000000000056
(gdb) x&var4
0x600184:       0x0000000000000012
(gdb) x/8gx &ptrs
0x60018c:       0x0000000000000000      0x0000000000000000
0x60019c:       0x0000000000000000      0x0000000000000000
0x6001ac:       0x0000000000000000      0x0000000000000000
0x6001bc:       0x0000002c00000000      0x0008000000000002
(gdb)
```

Pointer variables being initialized:



```
native process 25766 In: loop2
(gdb) x&var2
0x600174:       0x00000028
(gdb) x&var3
0x60017c:       0x00000056
(gdb) x&var4
0x600184:       0x00000012
(gdb) x/6gx &ptrs
0x60018c:       0x0000000000000000      0x0000000000000000
0x60019c:       0x0000000000000000      0x0000000000000000
0x6001ac:       0x0000000000000000      0x0000000000000000
(gdb) step
(gdb) x/6gx &ptrs
0x60018c:       0x000000000060016c      0x0000000000600174
0x60019c:       0x0000000000000000      0x0000000000000000
0x6001ac:       0x0000000000000000      0x0000000000000000
(gdb) step
loop2 () at linkedList.s:33
(gdb) x/6gx &ptrs
0x60018c:       0x000000000060016c      0x0000000000600174
0x60019c:       0x000000000060017c      0x0000000000600184
0x6001ac:       0x0000000000000000      0x0000000000000000
(gdb) 
```

Gdb register layout and data elements after operation:



```
lqqRegister group: generalqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
xrax            0x1      1                 rbx            0x56     86                    x
xrcx            0x60017c 6291836           rdx            0x60017c 6291836               x
xrsi            0x3      3                 rdi            0x4      4                     x
xrbp            0x0      0x0               rsp            0x7fffffffe1f0   0x7fffffffe1f0 x
xr8             0x0      0                 r9             0x0      0                     x
xr10            0x0      0                 r11            0x0      0                     x
xr12            0x0      0                 r13            0x0      0                     x
xr14            0x0      0                 r15            0x0      0                     x
xrip            0x40016a 0x40016a <exit+7> eflags         0x246    [ PF ZF IF ]          x
xcs             0x33     51                ss             0x2b     43                    x
xds             0x0      0                 es             0x0      0                     x
xfs             0x0      0                 gs             0x0      0                     x
x                                                                                        x
x                                                                                        x
x                                                                                        x
x                                                                                        x
x                                                                                        x
x                                                                                        x
x                                                                                        x
   lqqlinkedList.sqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
   x49                je exit                                                        x
   x50                jmp loop2                                                      x
   x51                                                                               x
   x52                                                                               x
   x53        swap:   movq %rcx, ptrs(,%rsi,8)                                       x
   x54                dec %rsi                                                       x
   x55                movq  %rdx, ptrs(,%rsi,8)                                      x
   x56                jmp loop                                                       x
   x57                                                                               x
   x58        exit:   movq  $1,%rax                                                  x
  >x59                int $0x80                                                      x
   x60                                                                               x
   x61                                                                               x
   x62                                                                               x
   x63                                                                               x
   x64                                                                               x
   x65                                                                               x
   x66                                                                               x
   x67                                                                               x
   x68                                                                               x
   x69                                                                               x
   mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj
native process 25756 In: exit                                          L59   PC: 0x40016a
outer () at linkedList.s:47
loop2 () at linkedList.s:33
loop () at linkedList.s:34
swap () at linkedList.s:53
loop () at linkedList.s:34
outer () at linkedList.s:47
exit () at linkedList.s:58
(gdb) x&var1
0x60016c:       0x0000000000000031
(gdb) x&var2
0x600174:       0x0000000000000028
(gdb) x&var3
0x60017c:       0x0000000000000056
(gdb) x&var4
0x600184:       0x0000000000000012
(gdb) x/8gx &ptrs
0x60018c:       0x0000000000600184      0x0000000000600174
0x60019c:       0x000000000060016c      0x000000000060017c
0x6001ac:       0x0000000000000000      0x0000000000000003
0x6001bc:       0x0000002c00000000      0x0008000000000002
(gdb) 
```

As you can see, the values in the variables haven't changed, but the pointer values have. This is how we implemented a version of linked list implementation.

10: IF IT DIDN'T WORK, WHY?

We were able to implement the baseline specifications and it worked properly. We were not able to implement all the enhancements that we wanted to, including merge sort and quick sort. These proved rather difficult in Assembly, but we have included the C code in our repository to show our efforts. We did have some trouble initially with the second to last element not sorting properly but were able to debug and figure out there was an error in the pointer manipulation in our Swap section.

11: REFERENCES

Introduction to Algorithms (for our bubble sort algorithm basis)

Professor's previous lab assignment code

Lecture slides (for gdb, assembly information, etc.)

Andy previous C++ implementation of bubble sort (for attempting C bubblesort)

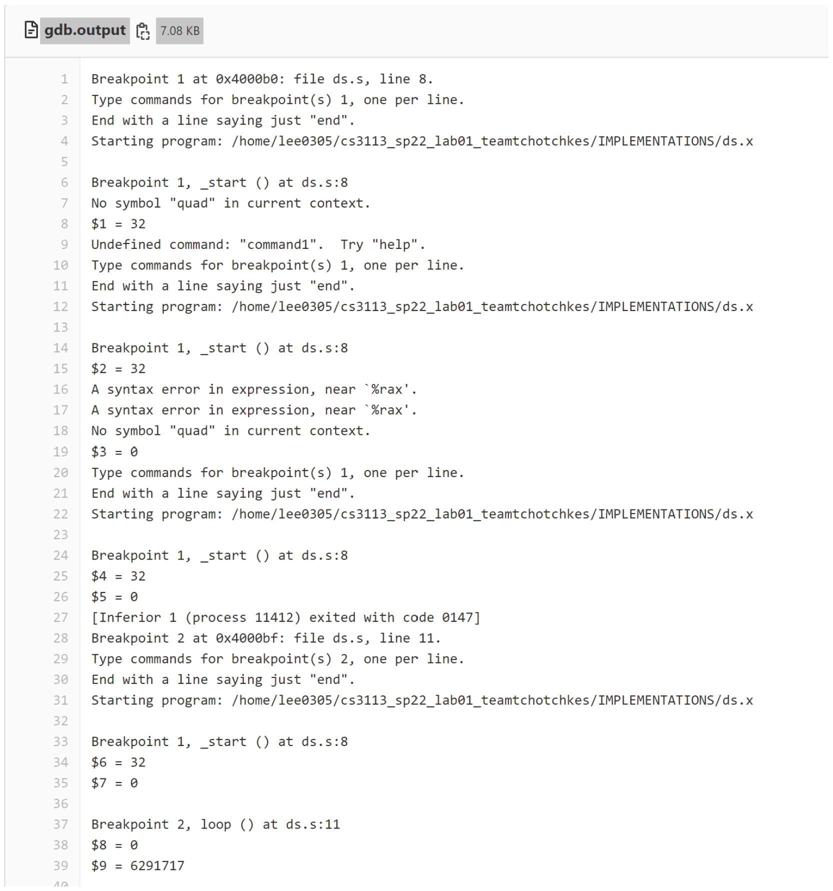
Geeksforgeeks

Stackoverflow

Youtube

12: ANYTHING ELSE?

As an enhancement to our implementation, we created a GDB script that shows the printout of our memory and values at each of the breakpoints we defined. It was very helpful in furthering our understanding of the functionality of the CPU and the operating system. It's rather long so for brevity we will just have a snippet of it here. Please see below:



gdb.output 7.08 KB

```
Breakpoint 1 at 0x4000b0: file ds.s, line 8.
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
Starting program: /home/lee0305/cs3113_sp22_lab01_teamtchotchkes/IMPLEMENTATIONS/ds.x

Breakpoint 1, _start () at ds.s:8
No symbol "quad" in current context.
$1 = 32
Undefined command: "command1".  Try "help".
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
Starting program: /home/lee0305/cs3113_sp22_lab01_teamtchotchkes/IMPLEMENTATIONS/ds.x

Breakpoint 1, _start () at ds.s:8
$2 = 32
A syntax error in expression, near `%rax'.
A syntax error in expression, near `%rax'.
No symbol "quad" in current context.
$3 = 0
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
Starting program: /home/lee0305/cs3113_sp22_lab01_teamtchotchkes/IMPLEMENTATIONS/ds.x

Breakpoint 1, _start () at ds.s:8
$4 = 32
$5 = 0
[Inferior 1 (process 11412) exited with code 0147]
Breakpoint 2 at 0x4000bf: file ds.s, line 11.
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
Starting program: /home/lee0305/cs3113_sp22_lab01_teamtchotchkes/IMPLEMENTATIONS/ds.x

Breakpoint 1, _start () at ds.s:8
$6 = 32
$7 = 0

Breakpoint 2, loop () at ds.s:11
$8 = 0
$9 = 6291717
```

The remainder of the script output can be found on our repository under Implementations. Also, we implemented bubble sort using array for our other enhancement, as a different way to approach the problem. We believe that this would be more efficient depending on the application, since there is no need to point to anything; for instance, if you can guarantee your input data, it is always going to be in a specific section of memory.