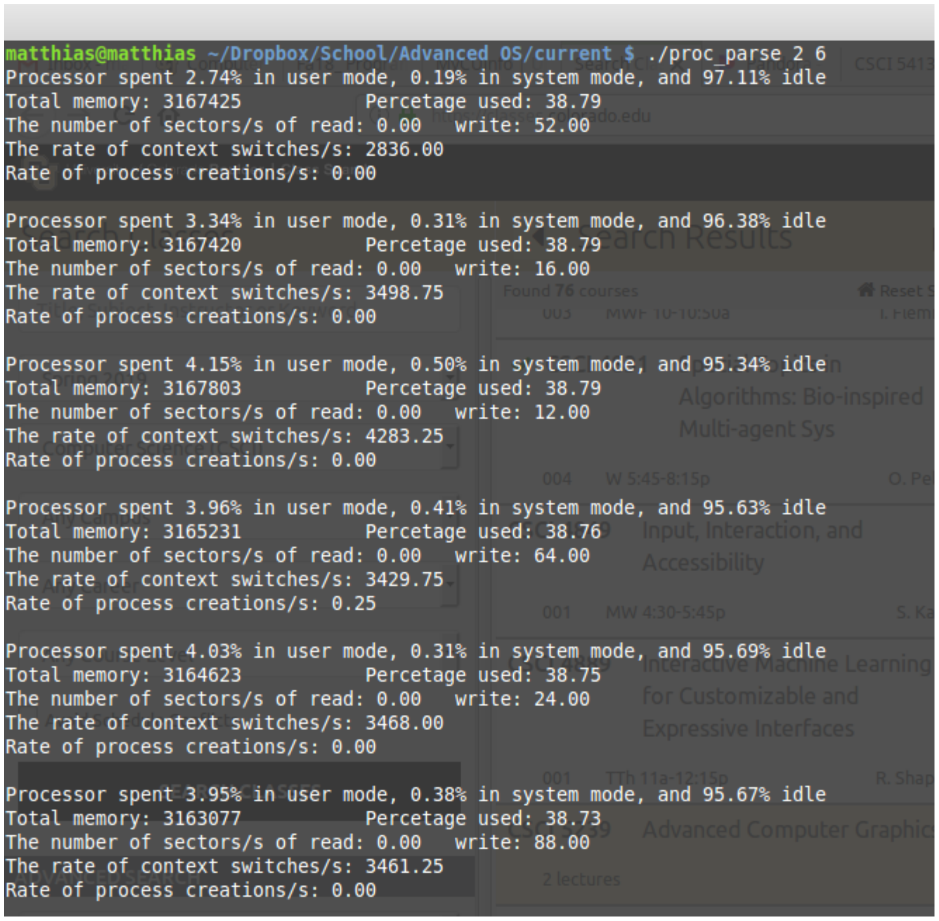
1. Here is a screenshot of my code running. All separate parts were individually tested on my desktop.



```
matthias@matthias ~/Dropbox/School/Advanced OS/current $ ./proc_parse 2 6
Processor spent 2.74% in user mode, 0.19% in system mode, and 97.11% idle
Total memory: 3167425           Percetage used: 38.79
The number of sectors/s of read: 0.00   write: 52.00
The rate of context switches/s: 2836.00
Rate of process creations/s: 0.00

Processor spent 3.34% in user mode, 0.31% in system mode, and 96.38% idle
Total memory: 3167420           Percetage used: 38.79
The number of sectors/s of read: 0.00   write: 16.00
The rate of context switches/s: 3498.75
Rate of process creations/s: 0.00

Processor spent 4.15% in user mode, 0.50% in system mode, and 95.34% idle
Total memory: 3167803           Percetage used: 38.79
The number of sectors/s of read: 0.00   write: 12.00
The rate of context switches/s: 4283.25
Rate of process creations/s: 0.00

Processor spent 3.96% in user mode, 0.41% in system mode, and 95.63% idle
Total memory: 3165231           Percetage used: 38.76
The number of sectors/s of read: 0.00   write: 64.00
The rate of context switches/s: 3429.75
Rate of process creations/s: 0.25

Processor spent 4.03% in user mode, 0.31% in system mode, and 95.69% idle
Total memory: 3164623           Percetage used: 38.75
The number of sectors/s of read: 0.00   write: 24.00
The rate of context switches/s: 3468.00
Rate of process creations/s: 0.00

Processor spent 3.95% in user mode, 0.38% in system mode, and 95.67% idle
Total memory: 3163077           Percetage used: 38.73
The number of sectors/s of read: 0.00   write: 88.00
The rate of context switches/s: 3461.25
Rate of process creations/s: 0.00
```

2. Answered with help from this article[1]

   (a) **The heap grows upward in memory.** We can see this in the `mallocer` program. To begin with, there is no heap, because malloc has not been called. Then, after we call malloc, we see that there is a heap. On my machine when I just ran this, by looking in /proc/pid/maps, it used the address spaces following:
   02112000-02133000 rw-p 00000000 00:00 0 [heap]
   So we can see that the heap is allocated from 02112000 to 02133000. However,

[1]https://blog.holbertonschool.com/hack-the-virtual-memory-malloc-the-heap-the-program-break/

when it is run the second time in the loop, allocating lots of space, we see the stack grow:

```
02112000-02154000 rw-p 00000000 00:00 0                    [heap]
```

We can see that the starting address, the lower bound, stays the same, but the upper address grows upward in memory.

The stack, however, grows down. Also by using maps, we can see when the call to the recursion is made many times, at the start of the function we see the following range for the stack:

```
7ffefc57b000-7ffefc59c000 rw-p 00000000 00:00 0                    [stack]
```

But at the deepest part of the recursion, right at the base case, we see the following range for the stack:

```
7ffefc28b000-7ffefc59c000 rw-p 00000000 00:00 0                    [stack]
```

So here we can see that the upper bound stays constant, but the lower bound stretches to make room for the many instances of function calls.

(b) We are referring to the following three segments:

```
00400000-00401000 r-xp 00000000 08:01 5636222                    /path_to_exe
00600000-00601000 r–p 00000000 08:01 5636222                     /path_to_exe
00601000-00602000 rw-p 00001000 08:01 5636222                    /path_to_exe
```

The first part, the lowest in memory, is the Code segment. This is the set of binary instructions that are actually run by the machine. This is why it has executable set in its permissions.

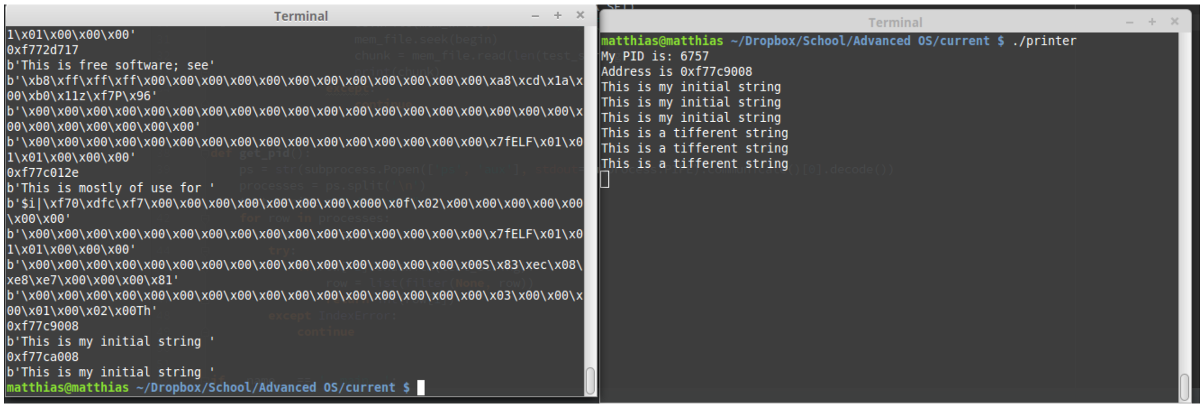
The second part is the read-only data - hence the permissions are read-only. This contains data such as the format strings for `printf` calls and other data that will never need to be written to.

The final part is the read-write segment. This contains the initialized and uninitialized data. Initialized data are things such as global or static variables, which would not appear in the stack with the other local variables. The unititalized data are any variables that are initialized to zero. This doesn't actually take up any space in the storage, but rather is a placeholder so that these variables can be used later. It is merely an optimization.

(c) The reason that the code is not stored at the very beginning of the heap is that before the actual data is a data structure that contains all of the metadata about the variable. How can we tell? Well, by printing out the memory address, we can see that the allocated space is always 0x10 bytes ahead of the beginnning of the address space for the heap. So, in this case, the pointer points to 0x22e9010, whereas /proc reports the heap starting at 022e9000, or 16 bytes back. When printing out the preceeding bytes, we see the occurance of 0400, which is the hexidecimal representation of 1024, the amount of space allocated for our variable. So, we can reasonably assume that that is the size, and that the other parts are other data about the object.

(d) The reason we do this is to prevent a heap overflow attack. If we put a bug in our program that allows a user to write past the limits of a buffer, then they can inject code into our heap and then execute it if they are clever. However, it is very difficult to go to that code if you are not able to predict where the code is going to be. So, the heap will not always be right after the program break - it will be assigned dynamically at run-time.

3. Here is a screenshot of my code running. The Python script is on top and the C program is on bottom.



```
1\x01\x00\x00\x00'
0xf772d717
b'This is free software; see'
b'\xb8\xff\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xa8\xcd\x1a\x
00\xb0\x11z\xf7P\x96'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x7fELF\x01\x0
1\x01\x00\x00\x00'
0xf77c012e
b'This is mostly of use for '
b'$i|\xf70\xdfc\xf7\x00\x00\x00\x00\x00\x00\x00\x000\x0f\x02\x00\x00\x00\x00\x00
\x00\x00'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x7fELF\x01\x0
1\x01\x00\x00\x00'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00S\x83\xec\x08\
xe8\xe7\x00\x00\x00\x81'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x
00\x01\x00\x02\x00Th'
0xf77c9008
b'This is my initial string '
0xf77ca008
b'This is my initial string '
matthias@matthias ~/Dropbox/School/Advanced OS/current $
```

```
matthias@matthias ~/Dropbox/School/Advanced OS/current $ ./printer
My PID is: 6757
Address is 0xf77c9008
This is my initial string
This is my initial string
This is my initial string
This is a tifferent string
This is a tifferent string
This is a tifferent string
```