Arlan Prado
Prof Chenhansa
cs 118-02

Lab 5

Purpose: The purpose of this assignment is to create a unique letter counter and letter search between two strings using functions and string instructions.

Process: The first thing to do is to label variables. 3 input variables: 2 strings being searched through and the string of characters that are being searched for. The output variables are the counts for unique between each string individually and for both strings and the counts of the characters for each and both strings.

```
12 .data
13 #INPUTS:
14 consC: .ascii "cly"                                  #consonants that we're searching for
15 strA: .ascii "Assembly Language is easy\0"
16 .equ lenA,(. - strA)
17 strB: .ascii "Chocolate is delicious\0"
18 .equ lenB,(. - strB)
19 #OUTPUTS:
20 A_countUni: .long 0             #StringA's Unique Character Count
21 B_countUni: .long 0             #StringB's Unique Character Count
22 AB_countUni: .long 0            #StringA and String B's Unique Character Count
23 A_countC1: .long 0              #stringA's count for consonant 1
24 B_countC1: .long 0              #stringB's count for consonant 1
25 AB_countC1: .long 0              #StringA and String B's count for consonant 1
26 A_countC2: .long 0              #stringA's count for consonant 2
27 B_countC2: .long 0              #stringA's count for consonant 2
28 AB_countC2: .long 0              #StringA and String B's count for consonant 2
29 A_countC3: .long 0              #stringA's count for consonant 3
30 B_countC3: .long 0              #stringA's count for consonant 3
31 AB_countC3: .long 0              #StringA and String B's count for consonant 3
32 .bss
33 .lcomm searchArr, 3                          #Uninitialized array holding the # of consonants
```

The first thing to do is to get all the characters in the strings into one case so it is easier to count characters. I chose to do it in lowercase since it is easier than doing it in uppercase. Changing it to lowercase only requires to mask all characters by 0x20 since masking lowercase characters already makes it lowercase. With uppercase, I'll have to identify the already uppercase letters and not to mask those.

To implement the lowercase function, I first put the string into register ESI. Then I push the length of the string into a the stack ESP. Next I call the function to lowercase the string. Push EBP then put copy the stack from ESP to EBP. Then I put the length from the stack into ECX and decrement the register to not have the null character masked. Then I take the one character at a time from the string up to the last character with MOVB. After masking it with or 0x20, I put the character back into its original spot and increment ECX. I jump out of this loop by checking if the register counting spaces (ECX)

and the register holding the length (EDX) is the same. After lowercasing all the characters, exit out the function and clear the stack.

```
34  .text
35  .globl _start, _convertLower, _letter, _consCount
36  _start:
37
38  leal strA, %esi
39  movl $lenA, %ecx
40  push %ecx
41  call _convertLower
42  addl $8, %esp
43
44  movl $0, %esi
45  leal strB, %esi
46  movl $lenB, %ecx
47  push %ecx
48  call _convertLower
49  addl $8, %esp
50  jmp next1
51
52  _convertLower:
53      pushl %ebp
54      movl %esp, %ebp
55      movl 8(%ebp), %edx                      #Length of string
56      decl %edx                               #Decrease length to not mask '/0'
57      movl $0, %ecx
58      movl $0, %ebx
59      lowerLetter:
60          movb (%esi, %ecx, 1), %bl           #Get the character
61          cmp %edx, %ecx
62          je done1
63          or $0x20, %bl                       #Mask the character to lowercase
64          movb %bl, (%esi, %ecx, 1)           #Put the masked character back in string
65          incl %ecx
66          jmp lowerLetter
67      done1:
68      pop %ebp
69      retl
70  next1:
```

After both strings are lowercased, the next function is to count unique characters. To do this I pushed the length of the string and the string address into the stack. Call the unique character count function then copy the address of the string into EDI and the length of the string into ECX. Next, I put the value of lowercase ‘a’ into a register. I would then scan every character of in the string with the single character register using REPNE SCASB. REPNE SCASB stops at the first instance of a character from EDI matching EAX. In turn it would stop ECX and ECX would not equal 0. Then I would increment the register counting unique characters. If it does equal 0, then reset ECX and the EDI to their original values. After each loop, the code would increment EAX to move onto the next letter.

The loop stops when the register that increments gets through the lowercase alphabet. Then it would quit the function and moves the number of characters found into its variable.
To get this function to count the unique characters between both strings, I added the lengths between both strings since it the memory is next to each other.

Function:

```
_letter:
    pushl %ebp
    movl %esp, %ebp
    movl $0x61, %eax
    movl $0, %edx

    unique:
        movl 8(%ebp), %edi              #Address of the string
        movl 12(%ebp), %ecx             #Length of the string
        cmp $'{', %al
        jae done2                       #If EAX/AL reaches the end of the alphabet, end function
        repne scasb                     #Scan through string until ECX is 0 or when character is found
        incl %eax
        cmp $0, %ecx                    #If ECX is not 0, character is found
        je unique
        incl %edx                       #Increment count of unique characters
        jmp unique
    done2:
        pop %ebp
        retl

next2:
```

Calls:

```
70  next1:
71  movl $lenA, %ecx
72  pushl %ecx
73  pushl $strA
74  calll _letter
75  addl $12, %esp
76  movl %edx, A_countUni
77
78  movl $lenB, %ecx
79  pushl %ecx
80  pushl $strB
81  calll _letter
82  addl $12, %esp
83  movl %edx, B_countUni
84
85  movl $lenA, %ecx
86  addl $lenB, %ecx
87  pushl %ecx
88  pushl $strA
89  calll _letter
90  addl $12, %esp
91  movl %edx, AB_countUni
92  jmp next2
93
```

Next is to count the amount of consonants for each string and between both strings. To do this, there is a third string is the characters being searched for and an uninitialized array of 3 holding the counts of each character. The function stack holds the string and length of the string being searched through. When the function is called: EBX is the counter for the array and the characters being searched, ESI is the string of consonants being searched for, and EDX is the array of counts of the consonants.

To go through the string and search through, the character of the third string is put into AL. ECX is the length of the string. Use REPNE SCASB through the string and stops when the character is found. If ECX is 0 after REPNE SCASB then the character searched for is not found and the character in AL is the next character being searched for. If ECX is not 0, increment at the index of the character that is being searched for. The uninitialized array's index would correspond to the index of the character in the third string. For example, if the character being searched for is in the first place of the third string, the index would be 0. And if the character being searched for at index 0 is found within the string, then the array of counts for the characters searched for would have index 0 be incremented. This would happen 3 times for the 3 consonants being searched for. After that, the function of exits. String A's count consonant variables would

take the numbers from the array directly, but the second call taking string B would result in both strings' counts, so I would subtract AB by A for the value to equal B.

Function:

```
158 _consCount:
159     pushl %ebp
160     movl %esp, %ebp
161
162     movl $-1, %ebx          #counter for characters searched
163     leal consC, %esi        #characters being searched for
164     movl $0, %eax
165     leal searchArr, %edx     #array counter for chars searched for
166     cons:
167         cmp $4, %ebx
168         je done3
169         movl 12(%ebp), %ecx     #length of string
170         movl 8(%ebp), %edi      #string
171         incl %ebx
172         consInside:
173             movb (%esi, %ebx, 1), %al
174             repne scasb
175             cmp $0, %ecx
176             je cons
177             incb (%edx, %ebx, 1)         #Increment the array counter at EBX
178             jmp consInside
179     done3:
180         pop %ebp
181         retl
182 next3:
```

Calls:

```
115  next2:
116  movl $lenA, %ecx
117  pushl %ecx
118  pushl $strA
119  calll _consCount
120  addl $12, %esp
121  movl $0, %ebx
122  movl $0, %edx
123  leal searchArr, %edx
124  movb (%edx, %ebx, 1), %al
125  movl %eax, A_countC1
126  incl %ebx
127  movb (%edx, %ebx, 1), %al
128  movl %eax, A_countC2
129  incl %ebx
130  movb (%edx, %ebx, 1), %al
131  movl %eax, A_countC3
132
133  movl $lenB, %ecx
134  pushl %ecx
135  pushl $strB
136  calll _consCount
137  addl $12, %esp
138  movl $0, %ebx
139  movl $0, %edx
140  leal searchArr, %edx
141  movb (%edx, %ebx, 1), %al
142  movl %eax, AB_countC1
143  subl A_countC1, %eax
144  movl %eax, B_countC1
145  incl %ebx
146  movb (%edx, %ebx, 1), %al
147  movl %eax, AB_countC2
148  subl A_countC2, %eax
149  movl %eax, B_countC2
150  incl %ebx
151  movb (%edx, %ebx, 1), %al
152  movl %eax, AB_countC3
153  subl A_countC3, %eax
154  movl %eax, B_countC3
```

Since all the values are now in its variables, exit the program normally.

```
182 next3:
183 end:
184 mov $1, %eax
185 mov $0, %ebx
186 int $0x80
187
```

Pitfalls:

The initial pitfall was trying to implement my idea of a 26 byte uninitialized array. It might have been a good idea but it requires a lot more work than the solution I currently have. It was a mistake to try to bite more than I could chew. So trying to implement the idea took more of my time than I should have, since in the end I chose the solution that got the lab done.

Another pitfall was discovering I assembled the program wrong and running the wrong executable. This was the problem:

```
debian@debian:~/lab5$ as apradfuns.s -g -o apradfuns.oA
debian@debian:~/lab5$ ld apradfuns.o -o apradfuns
debian@debian:~/lab5$ gdb apradfuns
```

It allowed me to make “apradfuns.oA” file and I did not catch it until I was looking through my process of running it when I could not figure out why the code is doing weird things. An example of a weird thing was the gdb is showing the line of code of moving 0 to ECX, but the register would only turn ECX into 0 2 lines later. Another example is when the registers kept giving me the same wrong result every time even though I kept changing the values inside the registers.

Possible improvements: An improvement I could have made to my code is for the first function to have its string pushed into the stack so I wouldn't have to set it to a register before calling the function. Another improvement is to make the code more flexible by having an uninitialized array with the size of 26 bytes to get the number of characters. The array would get all numbers of letters in the string. This would accomplish getting the consonant counts and getting unique characters. I believe I could have made my last two functions into 1 if I used that idea and the large array could get any letter asked for. Although, it might have been too complicated for the amount of time I was working on the lab for.