Q1 OS interfaces

15 Points

Write the C or pseudocode for the simple function that constructs the following pipeline that counts the number of `main()` strings in all the files in the current directory.

```
grep main -r ./* | wc -l
```

No files uploaded

Q2 Assembly

19 Points

Below is C and assembly code for the `strlen()` function from the xv6 operating system (i.e., `strlen()` computes the length of the string). In C strings are represented as continuous arrays of bytes (each character is a byte) that end with a `0` (or NULL) to designate the end of the string.

```
 96 int
 97 strlen(const char *s)
 98 {
 99   int n;
100
101   for(n = 0; s[n]; n++)
102     ;
103   return n;
104 }
```

```
199 00000200 <strlen>:
200  200:   55                      push   ebp
201  201:   89 e5                   mov    ebp,esp
202  203:   8b 4d 08                mov    ecx,DWORD PTR [ebp+0x8]
203  206:   80 39 00                cmp    BYTE PTR [ecx],0x0
204  209:   74 15                   je     220 <strlen+0x20>
205  20b:   31 d2                   xor    edx,edx
206  20d:   8d 76 00                lea    esi,[esi+0x0]
207  210:   83 c2 01                add    edx,0x1
208  213:   80 3c 11 00             cmp    BYTE PTR [ecx+edx*1],0x0
209  217:   89 d0                   mov    eax,edx
210  219:   75 f5                   jne    210 <strlen+0x10>
211  21b:   5d                      pop    ebp
212  21c:   c3                      ret
213  21d:   8d 76 00                lea    esi,[esi+0x0]
214  220:   31 c0                   xor    eax,eax
215  222:   5d                      pop    ebp
216  223:   c3                      ret
217  224:   8d b6 00 00 00 00       lea    esi,[esi+0x0]
218  22a:   8d bf 00 00 00 00       lea    edi,[edi+0x0]
```

Explain the role of ach line of assembly code (i.e., what function each line performs) (1 point for each line):

No files uploaded

Q3 Calling conventions

5 Points

Write the call site assembly code for the following function (assume that variable `a` is in register `EBX`, and you want to place the returned value back into `a`, i.e., back in `EBX`.

```
a = foo(a, 1, 2);
```

No files uploaded

Q4 Relocation

20 Points

In the following code of the `wc()` (word count) function from the xv6 operating system which lines will result in assembly code that would require relocation if loaded at a different memory address. Explain your answer. Assume that all functions that `wc()` uses are external, i.e., come from different object files. Xv6 is compiled without support for position independent code. (1 point for each non-trivial line)

```
 5 char buf[512];
 6
 7 void
 8 wc(int fd, char *name)
 9 {
10   int i, n;
11   int l, w, c, inword;
12
13   l = w = c = 0;
14   inword = 0;
15   while((n = read(fd, buf, sizeof(buf))) > 0){
16     for(i=0; i<n; i++){
17       c++;
18       if(buf[i] == '\n')
19         l++;
20       if(strchr(" \r\t\n\v", buf[i]))
21         inword = 0;
22       else if(!inword){
23         w++;
24         inword = 1;
25       }
26     }
27   }
28   if(n < 0){
29     printf(1, "wc: read error\n");
30     exit();
31   }
32   printf(1, "%d %d %d %s\n", l, w, c, name);
33 }
```

No files uploaded

Q5 Stacks
20 Points

Typically, the stack is implemented as a continuous region of memory that is pre-allocated by the operating system when the process (or a thread) start executing. This, however, has certain limitations: i.e., some programs need tiny stacks, and some large, so there is no size that fits all. As a system designer you decide to support stacks of variable length. Specifically, on entry to each function you want to check the size of the current stack and if it is less then a certain constant (which you can define) allocate more memory for the stack. Of course the stack is no longer a single continuous region of memory, but a collection of regions that are somehow linked together (it's your choice how to link them).

How do you need to change the prologue and epilogue of the function (i.e., what assembly code the compiler should generate on entry and return from a function for this idea to work). Explain your solution.

No files uploaded

Q6 Address translation

10 Points

Consider the following 32-bit x86 page table

`CR3` holds `0x0`.

The Page Directory Page is at physical address `0x0` (the flags are PTE_P (present), PTE_U (user-accessible), and PTE_W (writable):

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 2: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page is at physical address `0x00001000` (which is PPN `0x00001`):

```
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00006, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Another Page Table Page is at physical address `0x00002000` (which is PPN `0x00002`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00005, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

What physical address does the virtual (or if you like to be more specific linear) address `0x1000` translates to? Explain your answer.

No files uploaded

Q7 More page tables

10 Points

Using the same format for describing the page table as in the question above construct a page table that maps three pages at virtual addresses `0x8010 0000`, `0x8010 1000`, and `0x8010 2000` to physical addresses `0x0`, `0x8000_1000`, and `0xFFFF_F000`. Use x86-32, 4KB pages.

No files uploaded