Q1 OS interfaces

10 Points

Write a simple UNIX program, `tee` that reads from standard input and writes to standard output and a file specified in command line. For example, if invoked like this:

```
echo Hello | tee foobar.txt
```

"Hello" shows up on the screen and in foobar.txt

No files uploaded

Q2 Assembly

15 Points

Below is C and assembly code for the `strncpy()` (string copy) function from the xv6 operating system (i.e., `strpy()` copies one string into another). 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.

```
char*
strncpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  while(n-- > 0 && (*s++ = *t++) != 0)
    ;
  while(n-- > 0)
    *s++ = 0;
  return os;
}
```

```
00000190 <strncpy>:
 190:   55                      push   ebp
 191:   89 e5                   mov    ebp,esp
 193:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 196:   56                      push   esi
 197:   8b 4d 10                mov    ecx,DWORD PTR [ebp+0x10]
 19a:   53                      push   ebx
 19b:   8b 5d 0c                mov    ebx,DWORD PTR [ebp+0xc]
 19e:   89 c2                   mov    edx,eax
 1a0:   eb 19                   jmp    1bb <strncpy+0x2b>
 1a2:   8d b6 00 00 00 00       lea    esi,[esi+0x0]
 1a8:   83 c3 01                add    ebx,0x1
 1ab:   0f b6 4b ff             movzx  ecx,BYTE PTR [ebx-0x1]
 1af:   83 c2 01                add    edx,0x1
 1b2:   84 c9                   test   cl,cl
 1b4:   88 4a ff                mov    BYTE PTR [edx-0x1],cl
 1b7:   74 09                   je     1c2 <strncpy+0x32>
 1b9:   89 f1                   mov    ecx,esi
 1bb:   85 c9                   test   ecx,ecx
 1bd:   8d 71 ff                lea    esi,[ecx-0x1]
 1c0:   7f e6                   jg     1a8 <strncpy+0x18>
 1c2:   31 c9                   xor    ecx,ecx
 1c4:   85 f6                   test   esi,esi
 1c6:   7e 0f                   jle    1d7 <strncpy+0x47>
 1c8:   c6 04 0a 00             mov    BYTE PTR [edx+ecx*1],0x0
 1cc:   89 f3                   mov    ebx,esi
```

```
1ce:    83 c1 01                add    ecx,0x1
1d1:    29 cb                   sub    ebx,ecx
1d3:    85 db                   test   ebx,ebx
1d5:    7f f1                   jg     1c8 <strncpy+0x38>
1d7:    5b                      pop    ebx
1d8:    5e                      pop    esi
1d9:    5d                      pop    ebp
1da:    c3                      ret
1db:    90                      nop
1dc:    8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
```

Q2.1
5 Points

What happens if you replace instruction at address `190` with a `nop` instruction? (`nop` does nothing, i.e., it advances the instruction pointer to the next instruction but does not affect memory or registers).

No files uploaded

Q2.2
5 Points

Same as above, but now you put two `nop` instructions instead of instruction at address `1b7`

No files uploaded

Q2.3
5 Points

Same as above, but now you put `nop` instead of the instruction at address `1d7`

No files uploaded

Q3
20 Points

Below is a code snippet of the `cat()` function from the xv6 `cat` utility

```
#include "types.h"
#include "stat.h"
#include "user.h"

char buf[512];

void
cat(int fd)
{
  int n;

  while((n = read(fd, buf, sizeof(buf))) > 0) {
    if (write(1, buf, n) != n) {
      printf(1, "cat: write error\n");
      exit();
    }
  }
  if(n < 0){
    printf(1, "cat: read error\n");
    exit();
  }
}
```

Q3.1 Memory allocation
10 Points

For each variable used in the program above, explain where (stack/heap/data/bss section) this variable is allocated.

No files uploaded

Q3.2
10 Points

Which lines of the code above require relocation if loaded at a different memory address. Explain your answer (1 point for each non-trivial line)

No files uploaded

Q4
20 Points

Imagine you have an x86 machine which has all the same instructions as we discussed in class, besides that it does not have `call`, `ret`, `push` and `pop`. Imagine you're in control of the compiler and can generate any assembly code you like.

Q4.1
10 Points

Explain how can you support function invocations? Show an example of the assembly code that invokes the `int foobar(int a, int b, int c)` function.

No files uploaded

Q4.2
10 Points

How will you maintain the stack frame and return from the function? Show assembly code that maintains the stack frame inside `foobar()` and returns from it.

No files uploaded

Q5 Page tables
25 Points

Q5.1
10 Points

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

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

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Specify all virtual address ranges mapped by this page table (don't forget to mention the physical ranges to which each virtual range is mapped), e.g., `virt: [a - b] -> phys: [x - z]`

No files uploaded

Q5.2
15 Points

Using the same format for describing the page table as in the question above construct a page table that maps virtual addresses from 0x0 to 1MB (0x10_0000) and from 2GB (0x8000_0000) to 2GB + 1MB (0x8010_0000) to physical addresses from 0x0 to 1MB (0x10_0000). You're free to choose where your PTD and PT pages are located in physical memory.

No files uploaded