Q1 Operating system call interface  
25 Points

Q1.1 Simple programs with I/O redirection  
15 Points
Write a program in C, using the UNIX system call API that implements the following pipeline
cat < foo.txt | grep main | wc -l > bar.txt, i.e., it launches cat, grep, and wc with
appropriate I/O redirection and connects them with pipes. Limit your use of system calls to the
following, i.e., the xv6 subset:

```c
void fork()
void exec(path, args)
void wait()
int open(fd, R|W)
void close(fd)
void pipe(fd[2])
int dup(fd)
int read(fd, buf, n)
int write(fd, buf, n)
```

Q1.2  
5 Points
You are trying to implement a fork bomb, i.e., the program that forks endlessly until it exhaust the
memory of the system, with the code below. You’re running on a beefy machine that has a ton of
memory and can support a ton of processes. However each process is configured with a 4096
byte stack.

```c
void recursive_fork()
{
    fork();
    recursive_fork();
    return;
}

main()
{
    recursive_fork();
}
```

How many processes you will be able to create? Explain your answer.

Q1.3  
5 Points
Can you change the program above to really exhaust all available memory on the machine by
forking endlessly?
Q2 ASM and calling conventions

Consider the following assembly program:

```
foo:
    push ebp
    mov ebp, esp
    sub esp, 16
    mov DWORD PTR [ebp-4], 43
    mov edx, DWORD PTR [ebp+8]
    mov eax, DWORD PTR [ebp-4]
    add eax, edx
    leave
    ret

main:
    push ebp
    mov ebp, esp
    sub esp, 16
    mov DWORD PTR [ebp-4], 17
    push DWORD PTR [ebp-4]
    call foo
    add esp, 4
    mov DWORD PTR [ebp-4], eax
    add DWORD PTR [ebp-4], 23
    mov eax, DWORD PTR [ebp-4]
    leave
    ret
```

Q2.1 Assembly
10 Points

Explain the purpose of each line of the assembly code.

Q2.2 Stack
5 Points

Draw and upload a diagram/picture of the call-stack generated right before the add instruction inside foo (or use the text field to provide an ASCII drawing). Explain every value on the stack.

Q2.3 Return values
5 Points

What value is returned by main?

Q2.1
1 foo function label
2 save old frame pointer
3 sets new frame pointer
4 allocating space for local variables (one integer + 64bits alignment)
5 move 43 into local variable
6 move argument passed into edx
7 move local variable 43 into eax
8 add edx and eax into eax for return
9 deallocate local variable (esp=esp+16) pop old ebp into ebp register
10 pop return address into program counter register (pc)

Q2.2
stack. slots of 4 bytes
...
...
...-padding- <---- Stack Pointer
-p Lod Padding-
-p Lod Padding-
43
ebp (of main)
ret_addr (return to main:18)
17
...-padding-
...
...

Q2.3
main returns the integer 83.
Q3.1 Allocation of variables
5 Points
For each variable used in the program above, explain where (stack/heap/data section) this variable is allocated.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>.rodata</td>
</tr>
<tr>
<td>ac</td>
<td>stack</td>
</tr>
<tr>
<td>av</td>
<td>stack</td>
</tr>
<tr>
<td>world</td>
<td>stack</td>
</tr>
<tr>
<td>world</td>
<td>stack/data (size&lt;83; gcc uses stack)</td>
</tr>
<tr>
<td>str</td>
<td>stack</td>
</tr>
<tr>
<td>literal &quot;beautiful&quot;</td>
<td>.rodata</td>
</tr>
</tbody>
</table>

Q3.2 Relocation
5 Points
If the program is relocated in memory by the linker, which lines in the program will result in assembly instructions that require relocation? Sometimes a single line results in multiple relocations, as it gets translated to multiple assembly lines (please explain them all).

Q3.2
While linking, text, data, and bss may move independently. Other (external) objects such as libraries, also move independently. Any cross reference among them will potentially need relocation.

Change?:
5: NO. stays in .rodata. whoever uses it (line 11) will take care of where hello is.
6: NO. We are just declaring main(). It may change its location, but it is just a label that some external code will use with a "call" instruction. Their problem.
8: NO. the literal "world!" is allocated in the stack at run time.
9: YES. malloc() may be somewhere else and we are calling it.
10: YES. memcpy() (from a library) and "beautiful" (.rodata) may be somewhere else. str gets its value at runtime, so no fixing on it.
11: YES. printf() and hello (pointer to .rodata) may be somewhere else.
12: NO. It is a simple instruction.