Q1 Operating system 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:

```
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 Stack
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.

```
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(Q1.2) to really exhaust all available memory on the machine by forking endlessly?

Q2 ASM and calling conventions
16 Points

Consider the following assembly program:

```
1  foo:
2          push    ebp
3          mov     ebp, esp
4          sub     esp, 16
5          mov     DWORD PTR [ebp-4], 43
6          mov     edx, DWORD PTR [ebp+8]
7          mov     eax, DWORD PTR [ebp-4]
8          add     eax, edx
9          mov     esp, ebp
10         pop     ebp
11         ret
12 main:
13         push    ebp
14         mov     ebp, esp
15         sub     esp, 16
```

```
16      mov     DWORD PTR [ebp-4], 17
17      push    DWORD PTR [ebp-4]
18      call    foo
19      add     esp, 4
20      mov     DWORD PTR [ebp-4], eax
21      add     DWORD PTR [ebp-4], 23
22      mov     eax, DWORD PTR [ebp-4]
23      mov     esp, ebp
24      pop     ebp
25      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.

No files uploaded

Q2.3 Return values
1 Point

What value is returned by `main`?

Q3 Linking and loading
10 Points

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    char world[] = "world!";
    char *str = malloc(64);
     memcpy(str, "beautiful", 64);
     printf("%s %s %s\n", hello, str, world);
     return 0;
}
```

Q3.1 Allocation
5 Points

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

Q3.2 Rellocation
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).