Midterm 2021Fall

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:

```
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.

```
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?

Q1.1

```
1 int p1[2], p2[2]
2 int fdin, fdout
3 pipe(p1)

4 if(fork()){
5     fdin=open("foo.txt",R)
6     close(0)
7     dup(fdin)
8     close(fdin)
9     close(1)
10    dup(p1[1])
11    close(p1[0])
12    close(p1[1])
13    exec("cat", [])

14 }else{
15    pipe(p2)
16    if(fork()){
17        close(0)
18        dup(p1[0])
19        close(p1[0])
20        close(1)
21        dup(p2[1])
22        close(p1[1])
23        exec("grep", ["main"])

24    }else{
25        close(0)
26        close(p1[0])
27        close(p1[1])
28        dup(p2[0])
29        close(p2[0])
30        fdout=open("bar.txt",W)
31        close(1)
32        dup(fdout)
33        close(fdout)
34        exec("wc", ["-l"])
35    }
36 }
```

```
1 two pipes
2 two files to open.
3 init pipe shared between cat and grep.

4 fork. if 1_parent, take care of cat:
5   open foo.txt in read mode.
6   close my stdin (fd 0).
7   make fd 0 point to foo.txt.
8   close this fd (the file still has somebody
    pointing to it, so it doesn't go away.
9   close stdout (fd 1).
10  make fd 1 point to the write end of the pipe.
11  close this fd for the read end of the pipe.
12  close this fd for the write end of the pipe.
13  in and out ready. now execute cat without arguments.

14 if 1_child, take care of the rest.
15  init pipe shared by grep and wc.
16  fork. if 2_parent, take care of grep:
17    close stdin.
18    make fd 0 point to the read end of the first pipe.
19    close the unused copy of fd.
20    close stdout.
21    make fd 1 point to the write end of the second pipe.
22    close the unused copy of fd.
23    in and out ready. now execute grep with arguments.

24 if 2_child, take care of wc:
25  close stdin.
26  close unused pipe.
27  close unused pipe.
28  make fd 0 point to the read end of second pipe.
29  close unused copy of the fd.
30  open bar.txt in write mode.
31  close stdout
32  make fd 1 point to bar.txt.
33  close unused copy of the fd.
34  in and out ready. execute wc with arguments.
```

Q1.2

In this scenario, the problem is not how many processes the system can handle, but the recursive call itself that will fill the stack. The maximum number of processes is given by two components:

- Every time we call a function, at least two elements get pushed to the call stack of the process: old ebp, and return call. Assuming a 32bits machine, that is 8 bytes each time.

- Every time we fork, the two resulting processes have the exact same state of memory (particularly their stack).

This is the sequence:
(1) the first process call recursive_fork() (push 8 bytes)
(2) recursive_fork calls fork (push 8 bytes) but returns (pop 8 bytes)
(3) recursive_fork calls itself (push 8 bytes)
(4) repeat from point (2)

Every time we pass through (2), we create a new process. Everytime we pass through (3) we push 8 bytes. This creates a binary tree of forks, and the final # of processes is the leaves of the tree. The tree ends there because the processes don't have more space in their stack.

Then, the # of processes is = 2^((4096-8)/8) = 2^511.
^---- from the first call made in 1)

Q1.3

The previous case ends because the stacks are full. If we fork without filling the stack, then we can create processes until the system has no more memory. So, calling fork() in a "while(1) fork();" loop will do the trick, or forking and exec the same program too. Maybe other aproaches too. The idea is to create an infinite loop-like situation, and create one process in each iteration or recursion.

Q2 ASM and calling conventions

20 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          leave
10         ret
11 main:
12         push    ebp
13         mov     ebp, esp
14         sub     esp, 16
15         mov     DWORD PTR [ebp-4], 17
16         push    DWORD PTR [ebp-4]
17         call    foo
18         add     esp, 4
19         mov     DWORD PTR [ebp-4], eax
20         add     DWORD PTR [ebp-4], 23
21         mov     eax, DWORD PTR [ebp-4]
22         leave
23         ret
```

Q2.1 Assembly

10 Points

Explain the purpose of each line of the assembly code.

```
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)
   https://stackoverflow.com/questions/4175281/what-does-it-mean-to-align-the-stack
 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)

11 main function label
12 save old frame pointer
13 sets new frame pointer
14 allocate local variable + 64bits alignment
15 assign 17 to local variable
16 pass local variable as argument to upcoming function call
17 push return address to stack, and set pc to point to foo
18 discard the argument passed
19 assign returned value to local variable
20 add 23 to local variable
21 put local variable in eax for return
22 deallocate local variable (esp=esp+16) pop old ebp into ebp register
23 pop return address into program counter register (pc)
```

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.

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

Q2.3 Return values

5 Points

What value is returned by `main`?

```
Q2.3
main returns the integer 83.
```

Q3 Linking and loading

10 Points

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4
 5 char hello[] = "Hello";
 6 int main(int ac, char **av)
 7 {
 8     char world[] = "world!";
 9     char *str = malloc(64);
10     memcpy(str, "beautiful", 64);
11     printf("%s %s %s\n", hello, str, world);
12     return 0;
13 }
```

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.

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).

```
Q3.1
hello   (array var) : .rodata
          (content) : .rodata

ac                  : stack

av        (pointer) : stack
          (content) : unknown

world   (array var) : stack
world     (content) : stack/data. (size<83; gcc uses stack)

str       (pointer) : stack
          (content) : heap

literal "beautiful" : .rodata
```

```
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.
```