Questions 1)

What is the difference between environment variables and shell variables?

Environment variables can be used not only in the current shell but also in all child processes started by that shell.
Shell variables can only be used inside the current shell.
At first, a shell variable and an environment variable can look the same.
But if don't use export, the shell variable will not become an environment variable.
Also, if the shell variable changes later, that change does not affect the environment variable.

Questions 2)

When we use execve() to execute an external program xyz, we pass NULL in the third argument. How many environment variables will the process xyz has?

It is zero. If the envp array, which holds environment variables, is NULL, no environment variables are passed to the process.
execvp() and execlp() call execve() inside, but they use a global variable called environ, so they pass environment variables automatically.
However, execve() needs the environment variable array as a direct input.
If you give NULL as the third argument, the program runs with no environment variables.

Subtask1)

```
[04/02/25]seed@VM:~/.../Lab3$ printenv | grep HANYANG
[04/02/25]seed@VM:~/.../Lab3$ export HANYANG="2020033781"
[04/02/25]seed@VM:~/.../Lab3$ printenv | grep HANYANG
HANYANG=2020033781
[04/02/25]seed@VM:~/.../Lab3$ unset HANYANG
[04/02/25]seed@VM:~/.../Lab3$ printenv | grep HANYANG
[04/02/25]seed@VM:~/.../Lab3$
```

At first, the environment variable HANYANG does not exist.
After using the export command, HANYANG is added to the environment variables.
Then, after using the unset command, HANYANG is removed from the environment.
We can see this step-by-step in the picture using the printenv | grep HANYANG command.

Subtask2)

```
[04/02/25]seed@VM:~/.../Lab3$ gcc myprintenv.c
[04/02/25]seed@VM:~/.../Lab3$ ./a.out > child.txt
[04/02/25]seed@VM:~/.../Lab3$ gcc myprintenv.c
[04/02/25]seed@VM:~/.../Lab3$ ./a.out > parent.txt
[04/02/25]seed@VM:~/.../Lab3$ diff child.txt parent.txt
[04/02/25]seed@VM:~/.../Lab3$
```

There is no output from the diff command, which means the two files are exactly the same.
This shows that the child process inherits the environment variables from the parent process.

Subtask3)

```
[04/02/25]seed@VM:~/.../Lab3$ a.out 1
[04/02/25]seed@VM:~/.../Lab3$ a.out 2
AAA=aaa
BBB=bbb
[04/02/25]seed@VM:~/.../Lab3$ a.out 3
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1347,unix/VM:/tmp/.ICE-u
nix/1347
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=1312
GTK_MODULES=gail:atk-bridge
DBUS_STARTER_BUS_TYPE=session
PWD=/home/seed/Systemsecurity/Lab3
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
WINDOWPATH=2
```

The execve() function replaces the current process with a new program.
In this experiment, we run the env command to check which environment variables are passed.

If I run a.out 1, NULL is passed as the third argument of execve().
Because of this, no environment variables are passed to the new program, so the env command shows nothing.

If I run a.out 2, a new array with AAA=aaa and BBB=bbb is passed as the environment. So the env command only prints those two variables.

If I run a.out 3, the global variable environ, which holds the current process's environment, is passed. In this case, the env command shows all the original environment variables.

Task1) Environment Variable and Set-UID Programs

```
[04/02/25]seed@VM:~/.../Lab3$ gcc -o myenv myenv.c
[04/02/25]seed@VM:~/.../Lab3$ sudo chown root myenv
[04/02/25]seed@VM:~/.../Lab3$ sudo chmod 4755 myenv
[04/02/25]seed@VM:~/.../Lab3$ export PATH=/home/seed:$PATH
[04/02/25]seed@VM:~/.../Lab3$ export LD_LIBRARY_PATH=/home/seed
[04/02/25]seed@VM:~/.../Lab3$ export HANYANG=<2020033781>
bash: syntax error near unexpected token `2020033781'
[04/02/25]seed@VM:~/.../Lab3$ export HANYANG=2020033781
[04/02/25]seed@VM:~/.../Lab3$ myenv | grep HANYANG
HANYANG=2020033781
[04/02/25]seed@VM:~/.../Lab3$ myenv | grep PATH
WINDOWPATH=2
PATH=/home/seed:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.
[04/02/25]seed@VM:~/.../Lab3$ myenv | grep LD_LIBRARY_PATH
[04/02/25]seed@VM:~/.../Lab3$
```

I used export command to set three environment variables: PATH, LD_LIBRARY_PATH, and HANYANG.
Then I ran a program with Set-UID to check whether these environment variables are passed to the child process.

As a result, HANYANG and PATH were printed, but LD_LIBRARY_PATH was not.

This is because the Linux system removes some environment variables automatically for security reasons when running Set-UID programs.

In particular, LD_LIBRARY_PATH can change the location of shared libraries used by the program.
If a user changes it in a harmful way, it can cause serious security problems, such as gaining root access.
To prevent this kind of attack, Linux is designed to block dangerous variables like LD_LIBRARY_PATH when running Set-UID programs.

From this experiment, we can see that Set-UID programs are affected by environment variables,
but some variables are restricted by the Linux system for security.

Task2)

```
[04/02/25]seed@VM:~/.../Lab3$ sudo ln -s /bin/zsh
 /bin/sh
[04/02/25]seed@VM:~/.../Lab3$ export PATH=.:$PATH
[04/02/25]seed@VM:~/.../Lab3$ gcc -o ls attack.c
[04/02/25]seed@VM:~/.../Lab3$ sudo chown root ls
[04/02/25]seed@VM:~/.../Lab3$ sudo chmod 4755 ls
[04/02/25]seed@VM:~/.../Lab3$ ls
# 
```

The test was done using zsh, which does not block Set-UID for security.
The PATH variable was set with .: at the beginning.
This means the system will look in the current directory first when searching for commands.
So if a user runs a command, a fake program in the current folder can be run instead of the real one.

The attack.c file is a simple program that runs a shell.
After compiling it and setting the Set-UID bit, the shell will run with root

permission.
In this case, the program was named ls, and because . comes first in PATH, running ls actually runs the fake ls in the current folder, not the real one.

As a result, when the user runs ls, a root shell appears instead of a file list. This shows that if a user changes PATH and runs a fake Set-UID program, they can gain a root shell.

Task3)

```
[04/03/25]seed@VM:~/.../Lab3$ gcc -o myprog myprog.c
[04/03/25]seed@VM:~/.../Lab3$ ./myprog
I am not sleeping!
[04/03/25]seed@VM:~/.../Lab3$ sudo chown root myprog
[04/03/25]seed@VM:~/.../Lab3$ sudo chmod 4755 myprog
[04/03/25]seed@VM:~/.../Lab3$ ./myprog
[04/03/25]seed@VM:~/.../Lab3$ sudo su
root@VM:/home/seed/Systemsecurity/Lab3# export LD_PRELOAD=./libmylib.so.1
root@VM:/home/seed/Systemsecurity/Lab3# ./myprog
I am not sleeping!
root@VM:/home/seed/Systemsecurity/Lab3#
```

$ gcc -fPIC -g -c mylib.c

$ gcc -shared -o libmylib.so.1 mylib.o -lc

$ export LD_PRELOAD=./libmylib.so.1

These commands create an object file (mylib.o) and a shared library file (libmylib.so.1).
Then, by setting LD_PRELOAD, we tell the system to load our custom library before others when a program runs.

When we ran myprog the first time, the output was I am not sleeping!. This means the program used the fake sleep function from our custom library instead of the original one.

Next, we changed the owner of myprog to root and set the Set-UID permission.
After doing this, when we ran the program again, the real sleep function was used.

This happened because Linux has a security rule: when a Set-UID program runs and the EUID (effective user ID) is different from the RUID (real user ID), the system ignores LD_PRELOAD to prevent attacks.

Later, we used sudo su to become the root user and ran myprog again. This time, I am not sleeping! was printed again.
This is because both EUID and RUID were root, so the system allowed LD_PRELOAD to work.