1. What form of exec() function you used? Why?

execvp(array[]) to execute ls-f and execvp(array[])) to execute nl. execvp() allows to pass vector/array as a parameter and inside of this array we can put our ls command and -f command and for another execvp() pass array for nl command and we have to put NULL at the end of array to mark end of array.
System call exec() is used it **replace** entire process specified in parentheses. When we run system call exec() PID stay the same and replace content of one process with another process. So it is not creating new one but replacing it. So after that place where exec() was called, everything after will be replaced but left content before exec() being called.

2. How many times you used fork? Why?
2 times. One to create one child and second one to create second child.
The fork() system call is used to create **separate, duplicate process**. gcc is compiler that compile our c file. It will generated compiled code with name a.out, which is executable file. Now we need to run this file in order to get output ./a.out
If we put system call fork() inside our c file it creates it is own child process. It means this process was executed 2 times, once by parent and another time by child. These 2 processes will have different PID because by definition it create separate duplicate process. If we run 3 system calls

fork() it will create 8 processes because on first call it creates 2 (parent and child) on second call it creates process for each parent and child, so now is 4 processes and if run fork() again it add again each process to all of them so now total is 8. Name a.out is default name. If we run several files it will override a.out, in order to prevent this we have to rename it by using command -o <nameOfFile>. Int fork() return value 0 for child, this way we can differentiate between parent and child and use in code by using 0 value for child and if it is value not 0, then it is parent. In order to prevent 2 process being created after fork() system call we can use if condition statement like !0 it means we are in parent and inside code block put another fork() so it will create 3 processes instead of 4.

3. How many pipes this assignment required? Why?

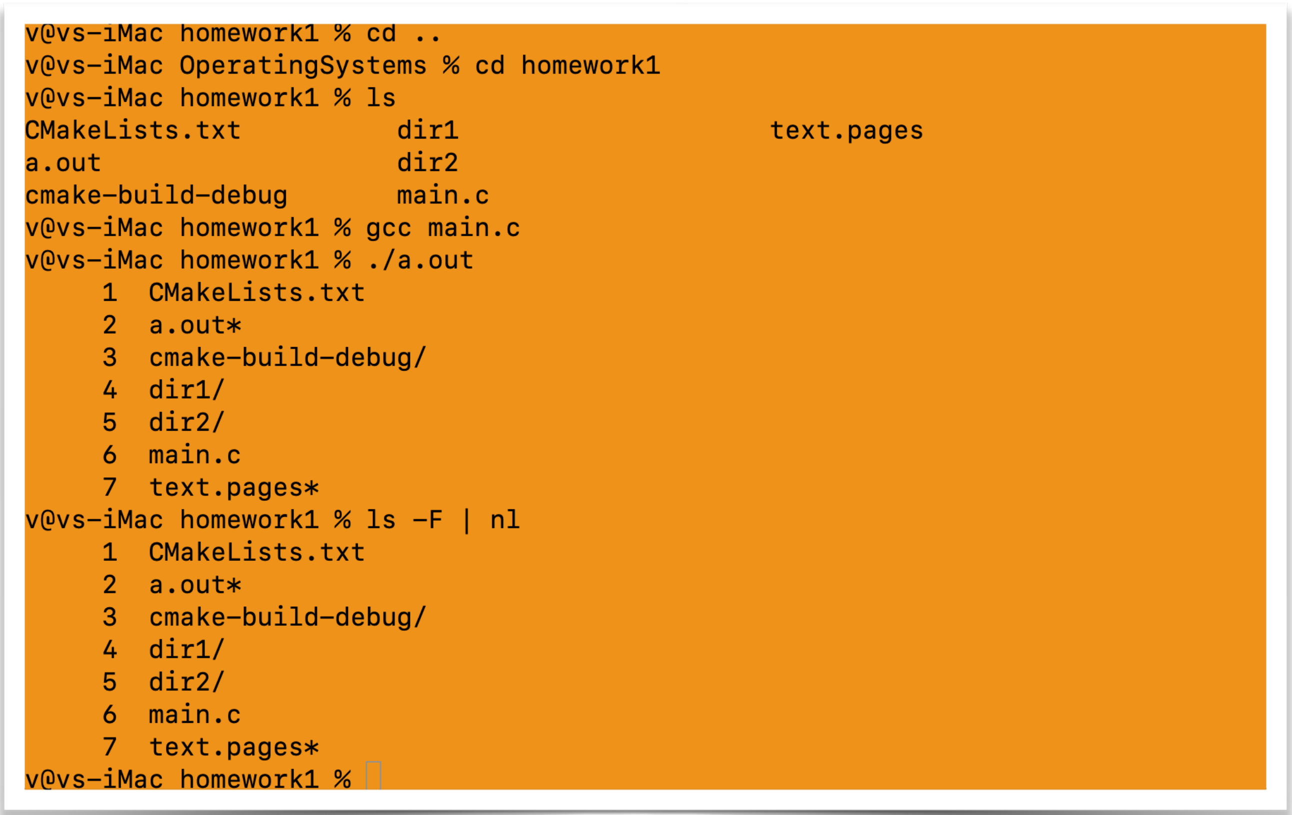
1 pipe it allows to read and write to it.
Pipe is like a buffer, saved in memory and we can write in it and read from it. pipe(fd) takes an array of 2 integers (int fd[2], like file descriptors like a key to access the file where we want to read or write data to. It is recommend if pipe return 0 it is successful, -1 is not : if (pipe(fd)==-1) printf("Pipe was not been able to open"). Why to have 2 file descriptors because word pipe as name imply to have 2 ends, so those 2 ends stored in int fd[2], where one end is fd[0] - read , where we can read from and other end fd[1] - write, it is where we can write data. For example write(fd[1] ,

second parameter where in memory, can be pointer and third parameter what we writing and then we can close this file descriptor close(fd[1]), that nobody can write to it. Once we fork() our file descriptors get inherited, as well need to close read end. And we can read(fd[0], from where and what). After we fork it creates parent and child processes therefore it will be 2 file descriptors with read and write each so total of 4 close(). It is good habit to check with help of "if" if == -1 another words checking for errors for every write(), read() or==0 means reach end of file.

4. What form of wait() you used? How many times?

2 times. It used for parent to wait while both children finish execution.

System call wait() is how to wait when process is finished. If we fork() it creates parent and child processes and if below we have code each process (parent & child) will executed that code in chaotic order. Sometimes it will run parent first, other time child first. It is where wait function comes in. Wait() says stop execution until child finish its execution and then do parent execution



```
v@vs-iMac homework1 % cd ..
v@vs-iMac OperatingSystems % cd homework1
v@vs-iMac homework1 % ls
CMakeLists.txt          dir1                    text.pages
a.out                   dir2
cmake-build-debug       main.c
v@vs-iMac homework1 % gcc main.c
v@vs-iMac homework1 % ./a.out
     1  CMakeLists.txt
     2  a.out*
     3  cmake-build-debug/
     4  dir1/
     5  dir2/
     6  main.c
     7  text.pages*
v@vs-iMac homework1 % ls -F | nl
     1  CMakeLists.txt
     2  a.out*
     3  cmake-build-debug/
     4  dir1/
     5  dir2/
     6  main.c
     7  text.pages*
v@vs-iMac homework1 % 
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    char *myargs[3]; // Define Arg (ls -F)
    char *myargs2[2]; // Define Second Arg (nl)

    int mypipe[2]; // Creating Array For Pipe

    if (pipe(mypipe) == -1){ // Initizing Pipe and Testing For Fails
        fprintf(stderr, "Pipe Failed");
        return 1;
    }

    int pid = fork(); // Creates 1 Child.
    if (pid < 0){ // Test For Fork Failed
        fprintf(stderr, "Fork Failed");
        return 1;
    }

    if (pid > 0) {
        // Parent
        //close(mypipe[0]); // Close Reading
        //close(mypipe[1]); // Close Writing
        //printf("Parent \n");
        int pid2 = fork(); //Creates Another Child

        if (pid2 < 0){ // Test For Fork Failed
            fprintf(stderr, "Fork Failed");
            return 1;
        }

        else if (pid2 > 0){
            // Parent
            wait(NULL); // Wait For First Child Process to Finish
            close(mypipe[1]); // Close Writing
            wait(NULL); // Wait For Second Child Process to Finish
            //printf("Done");
        }

        else if (pid2 == 0){ // Second Child from the Parent.
            // First Child
            //printf("Child1 \n");
            close(1); // Close stdout
            close(mypipe[0]); // Close Reading, Don't Need It
            myargs[0] = "ls";   // Perform ls
            myargs[1] = "-F"; // Perform -F
            myargs[2] = NULL; // Marks End of Array
            dup(mypipe[1]); // Making stdout the Same as mypipe[1]
            //write(mypipe[1], myargs, sizeof(myargs)); // Write to Pipe
```

```c
            //close(mypipe[1]); //Close Writing
            execvp(myargs[0], myargs); // Execute ls -F
            //execlp("ls","ls",NULL);
            printf("This Shouldn't Run \n"); // In Case of Failure
        }
        else {
            printf("This Shouldn't Run \n"); // In Case of Failure
        }
    }
    else if (pid == 0){
        // Second Child
        //printf("Child2 \n");
        //close(0); // Close stdin
        //close(mypipe[1]); // Close Writing, Don't Need It
        //dup(mypipe[0]); // Making stdin the Same as MyPipe[0]
        //read(mypipe[0], myargs, sizeof(myargs)); // Read the Pipe
        //printf("Back to Parent \n");
        close(0); // Close stdin
        close(mypipe[1]); // Close Writing, Don't Need It
        dup(mypipe[0]); // Making stdin the Same as MyPipe[0], which was taken from the first child
        close(mypipe[0]); // Close Reading, Already Read from dup, could also run without it
        myargs2[0] = "nl"; // Perform nl
        myargs2[1] = NULL; // Marks End of Array
        execvp(myargs2[0], myargs2); // Execute nl
        //close(mypipe[0]); // Close Reading
        //printf("%s \n", myargs[0]);
        //printf("%s \n", myargs[1]);
        //execlp("nl","nl",NULL);
        printf("This Shouldn't Run \n"); // In Case of Failure
    }
    else{
        printf("This Shouldn't Run \n"); // In Case of Failure
    }
    return 0;
}
```