**Cpt 5**

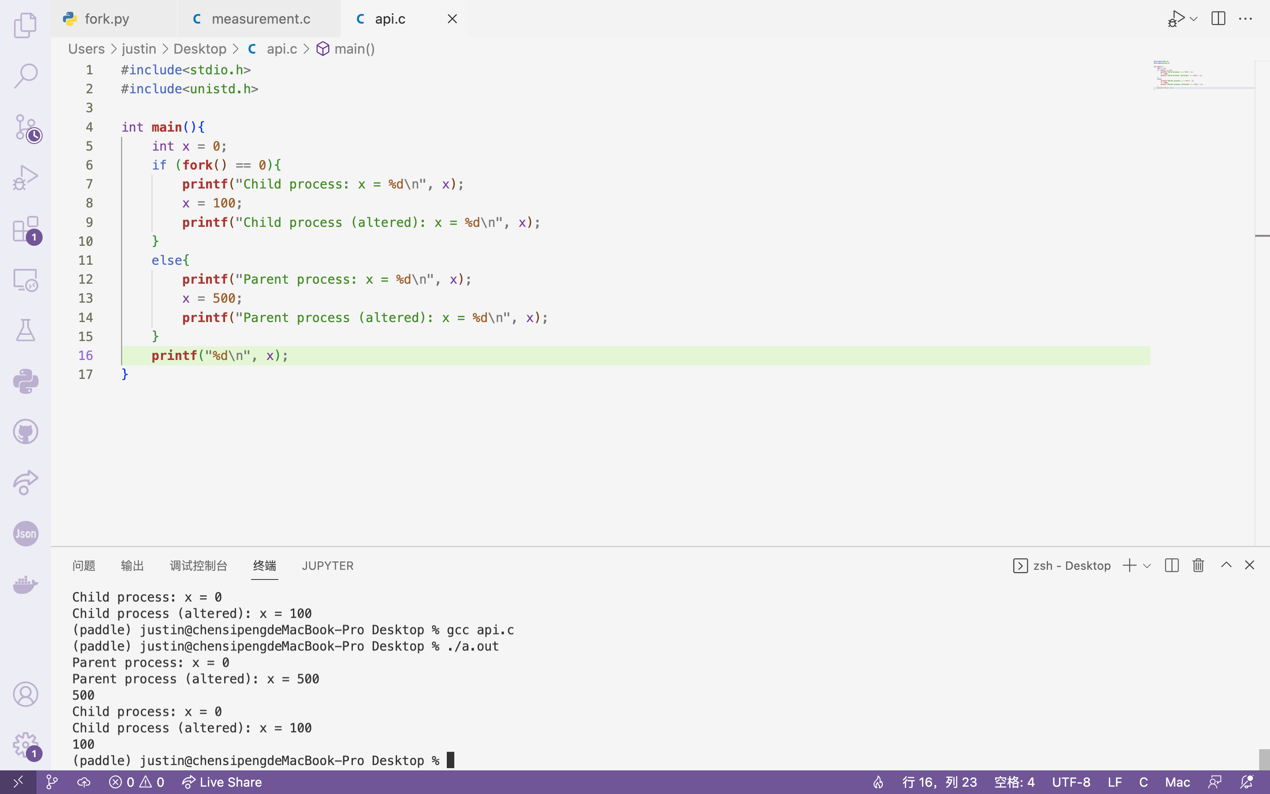
**Homework (code)**

1. **Write a program that calls fork(). Before calling fork(), have the main process access a variable (e.g., x) and set its value to some- thing (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x?**

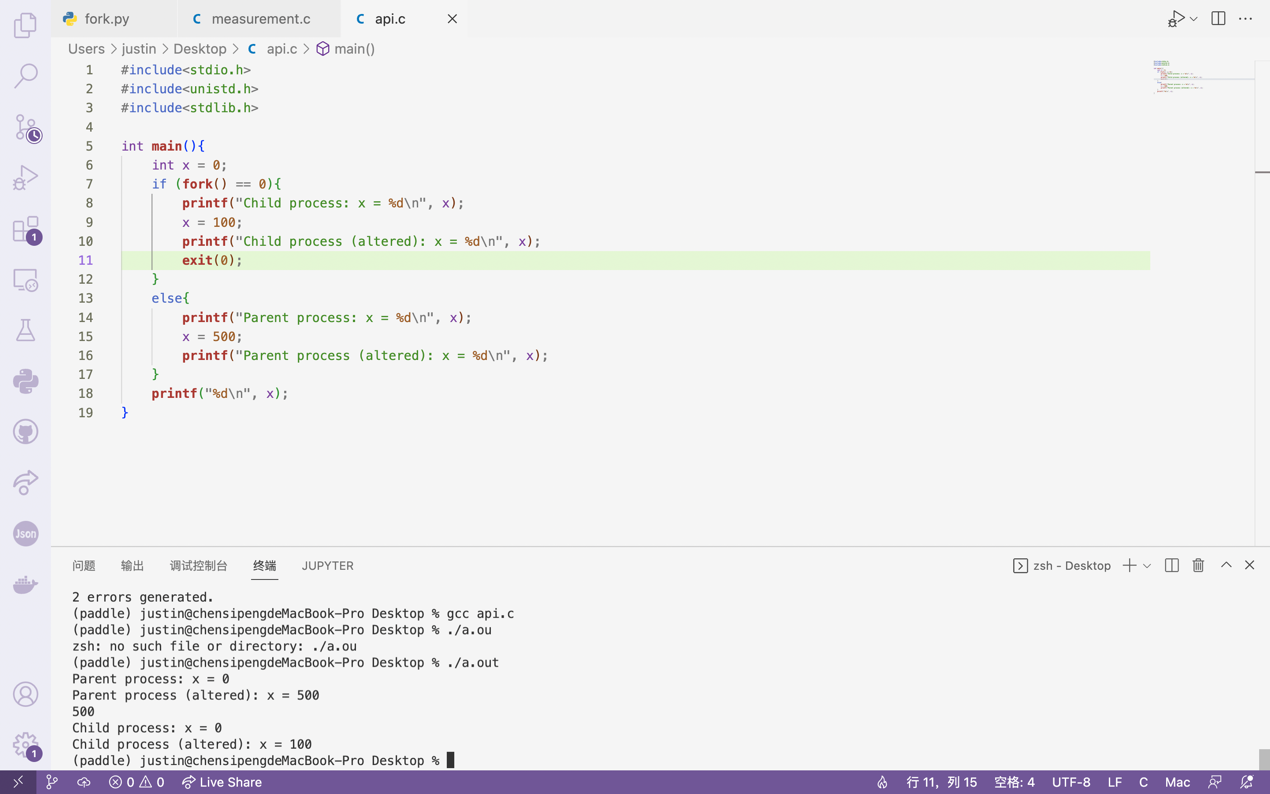
图形用户界面, 文本, 应用程序

描述已自动生成

Set variable x to 0 and fork out a child process. We can see from print that value of x is 0 in both process.

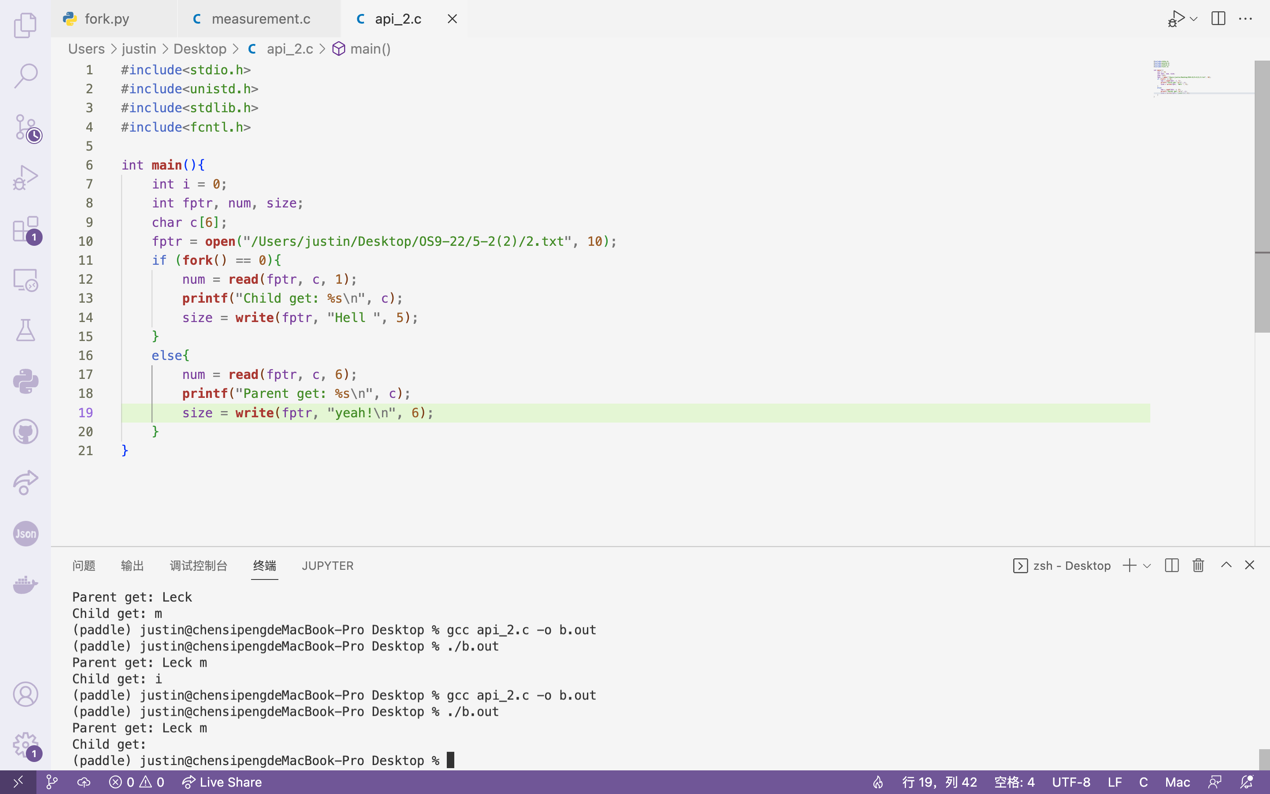


Change value of x in both process and print. The outcome shows that value of x in both process are independent from each other, which means if we change the value of x to 500 in parent process and 100 in child process, as is shown in the screenshot, the print outcome would be “Parent process (altered): 500” and “Child process (altered): 100”

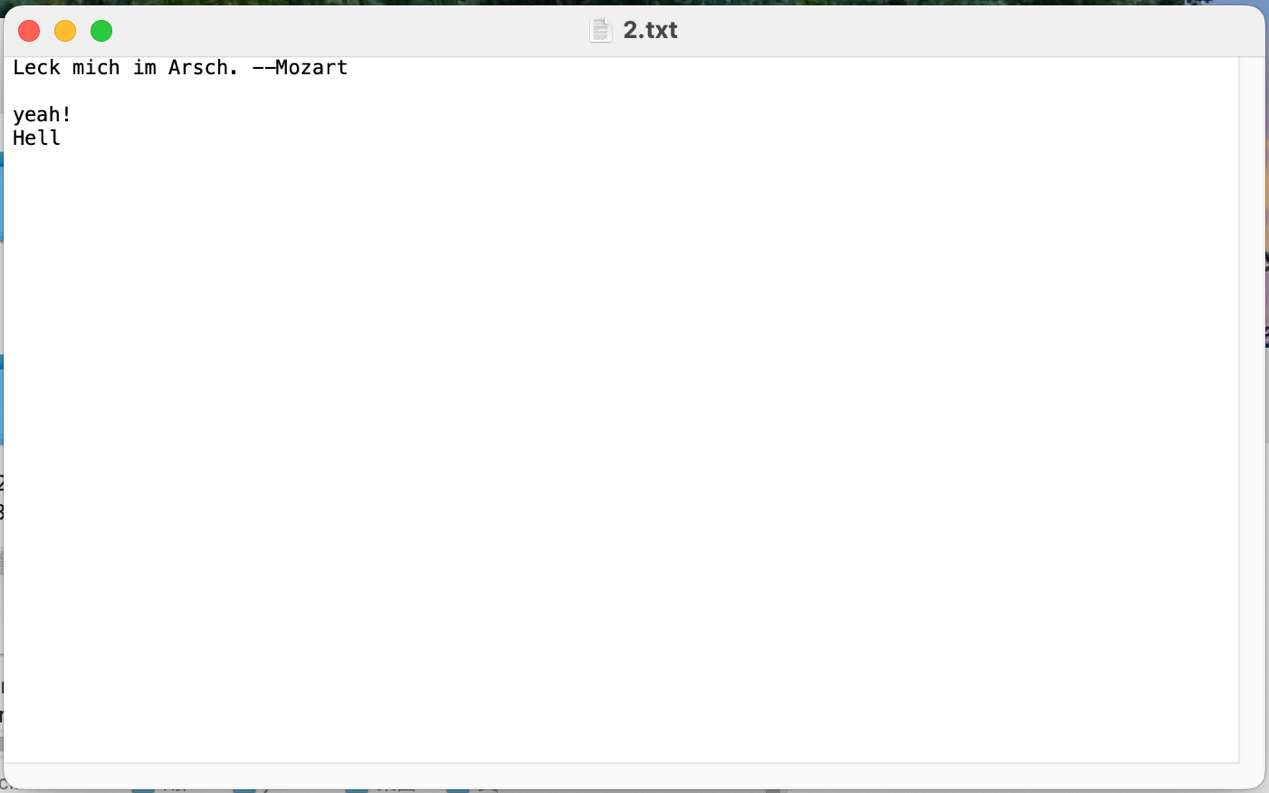


However, if we add “exit(0)” at the end of the child process, terminal no longer prints out the final result of the child process “100”, which is because child process exits after printing “Child process (altered): 100”. It can also be seen from the output that after child process exited, the variables in parent process will remain intact.

1. **Write a program that opens a file (with the open() system call) and then calls fork() to create a new process. Can both the child and parent access the file descriptor returned by open()? What happens when they are writing to the file concurrently, i.e., at the same time?**

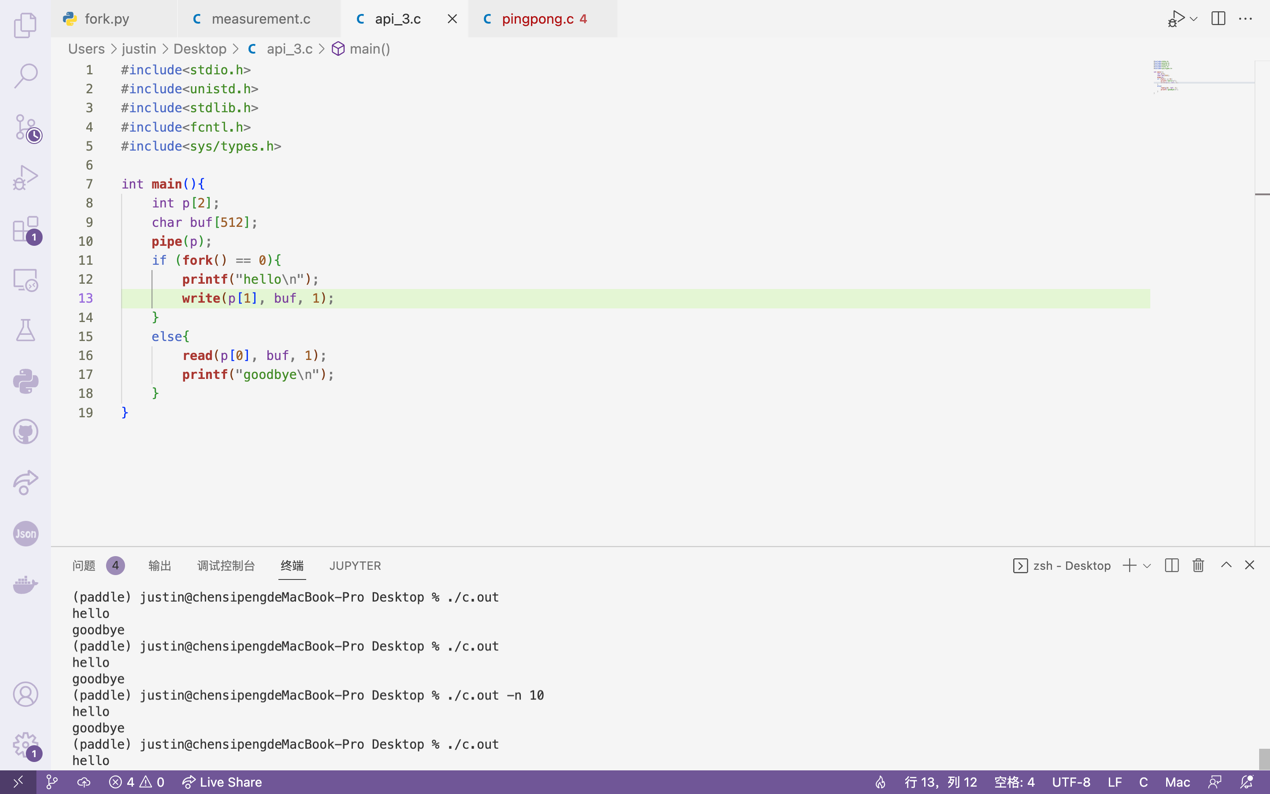
****

From the screenshot above we can see that both process can access the file indicated by the file descriptor fine.

****

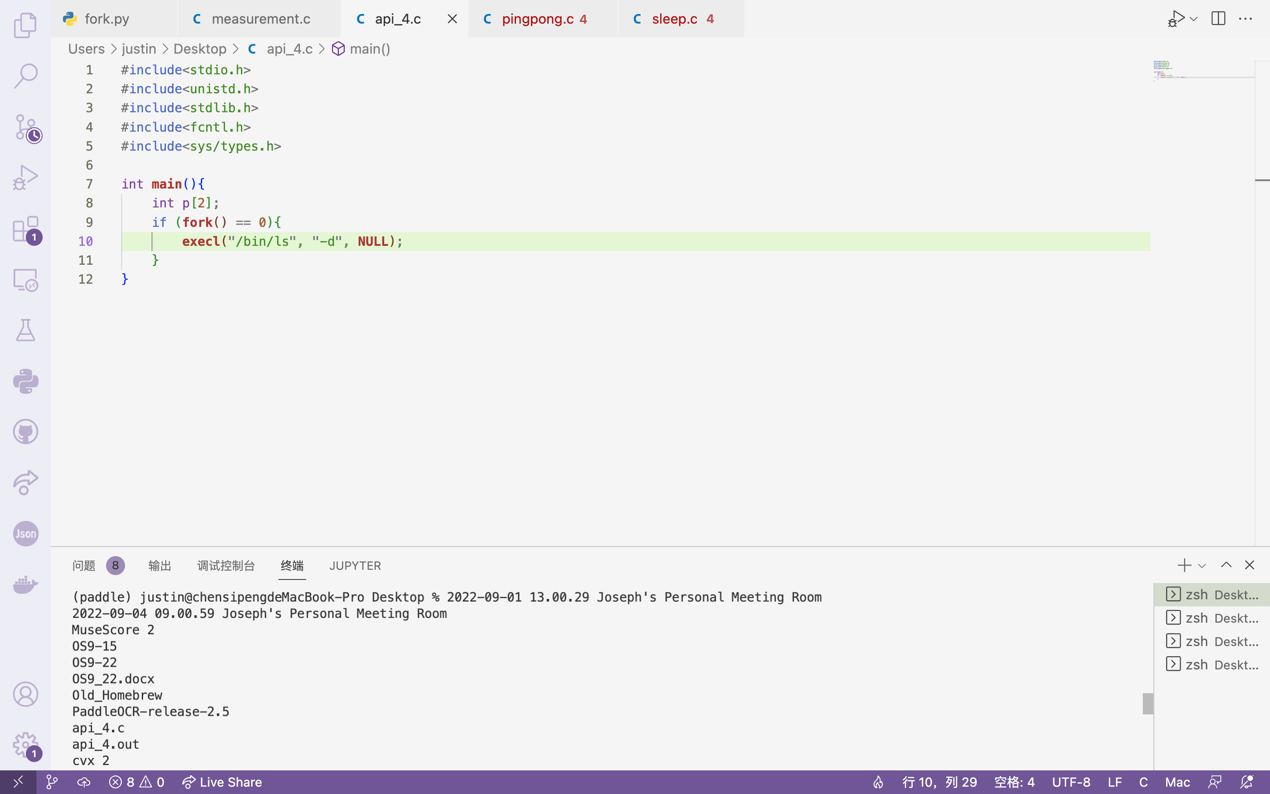
The outcome of “write” is coherent with the print outcome regarding the order of execution of child and parent process. As for the exact order, though, it’s simply random.

1. **Write another program using fork(). The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this without calling wait() in the parent?**

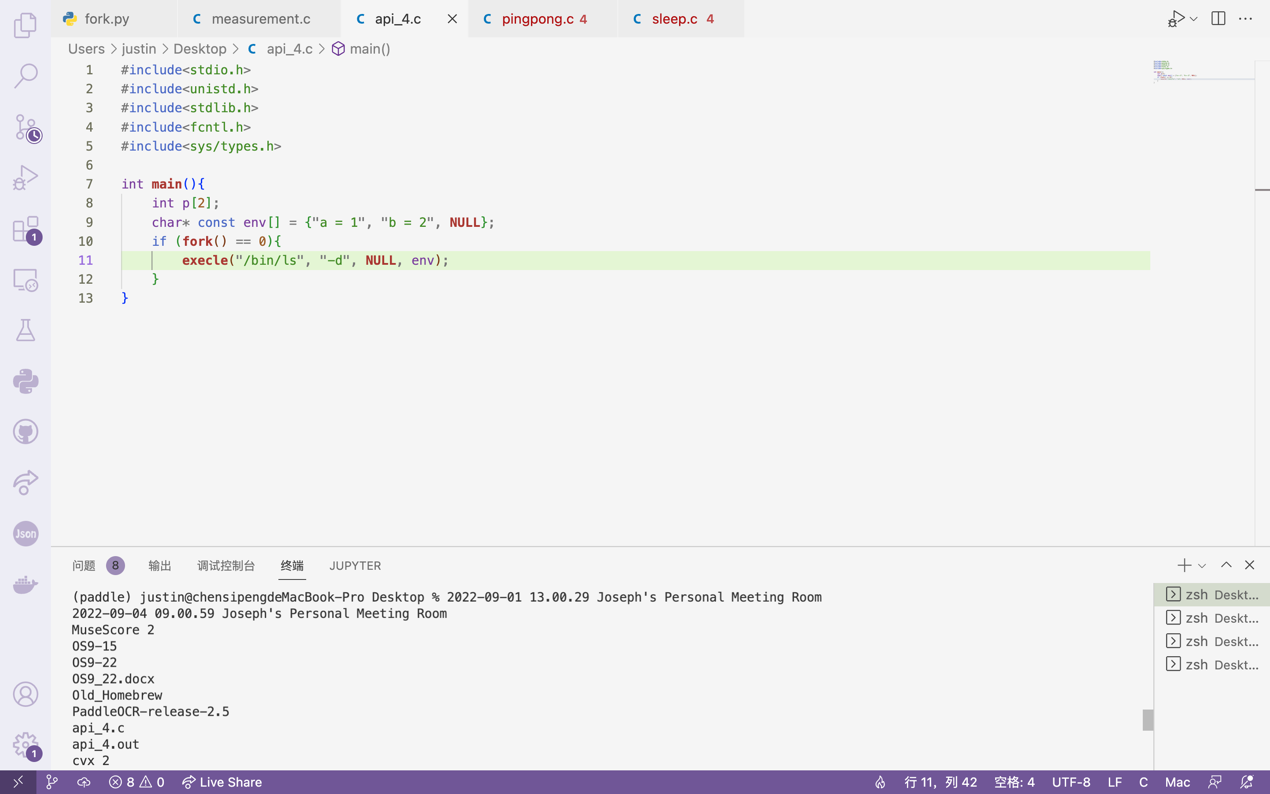


This would be quite simple if we can use wait(). As for the non-waiting version, we can achieve the same result with the help of pipe. By creating a pipe before forking out the child process and letting the child process send out a char after printing “hello” and letting the parent process read that char before printing “goodbye”, we can ensure that parent process always prints after child process has finished printing. It can be seen from the screenshot above that multiple trials have shown that this method indeed ensures the order of execution of parent and child process.

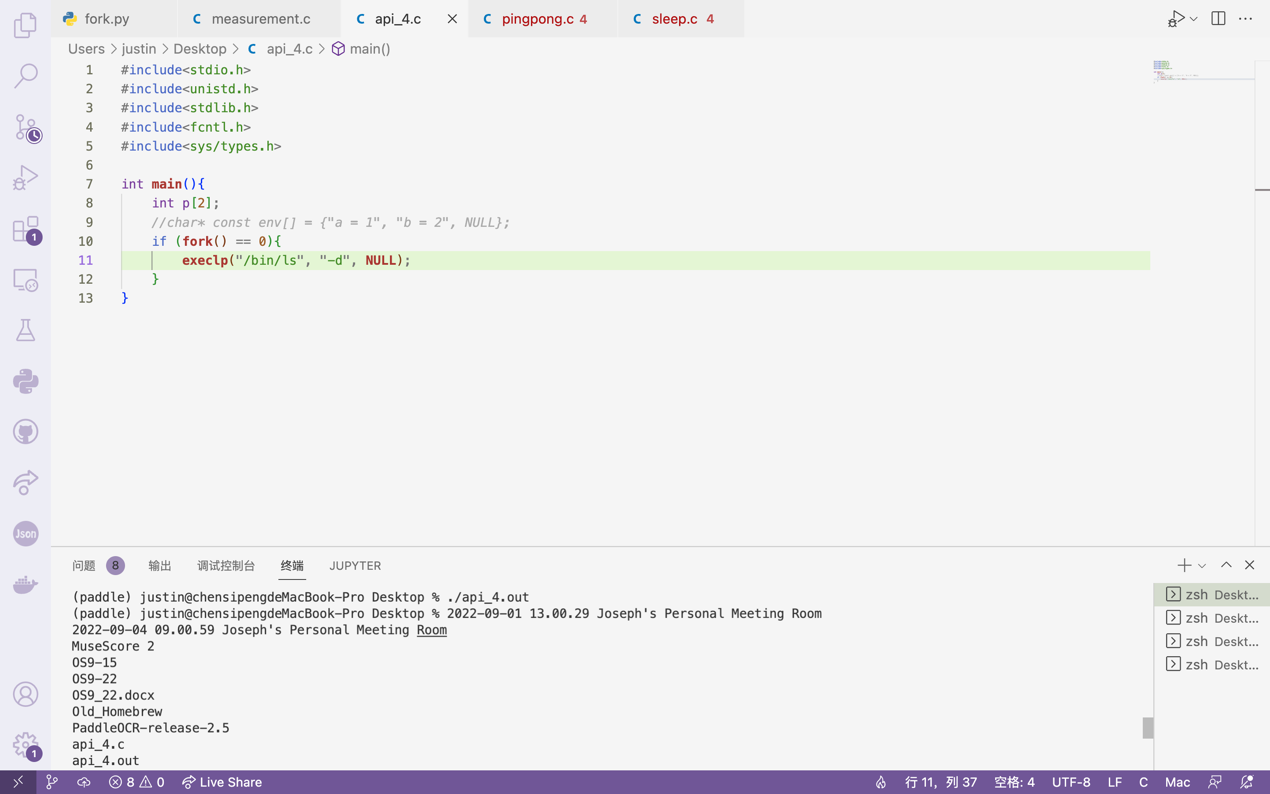
1. **Write a program that calls fork() and then calls some form of exec() to run the program /bin/ls. See if you can try all of the variants of exec(), including (on Linux) execl(), execle(), execlp(), execv(), execvp(), and execvpe(). Why do you think there are so many variants of the same basic call?**



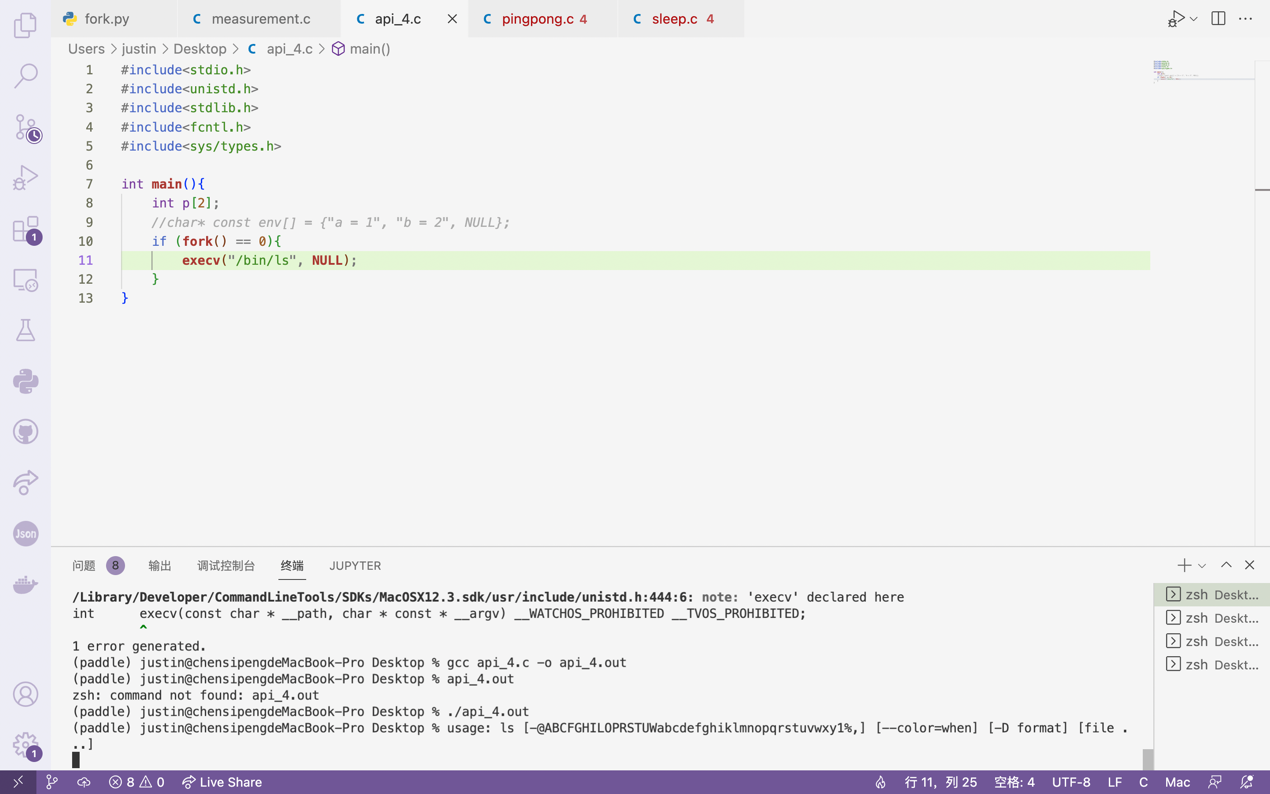
Execl() is called with the path of the file to be executed and a series of arguments which ends with a NULL.



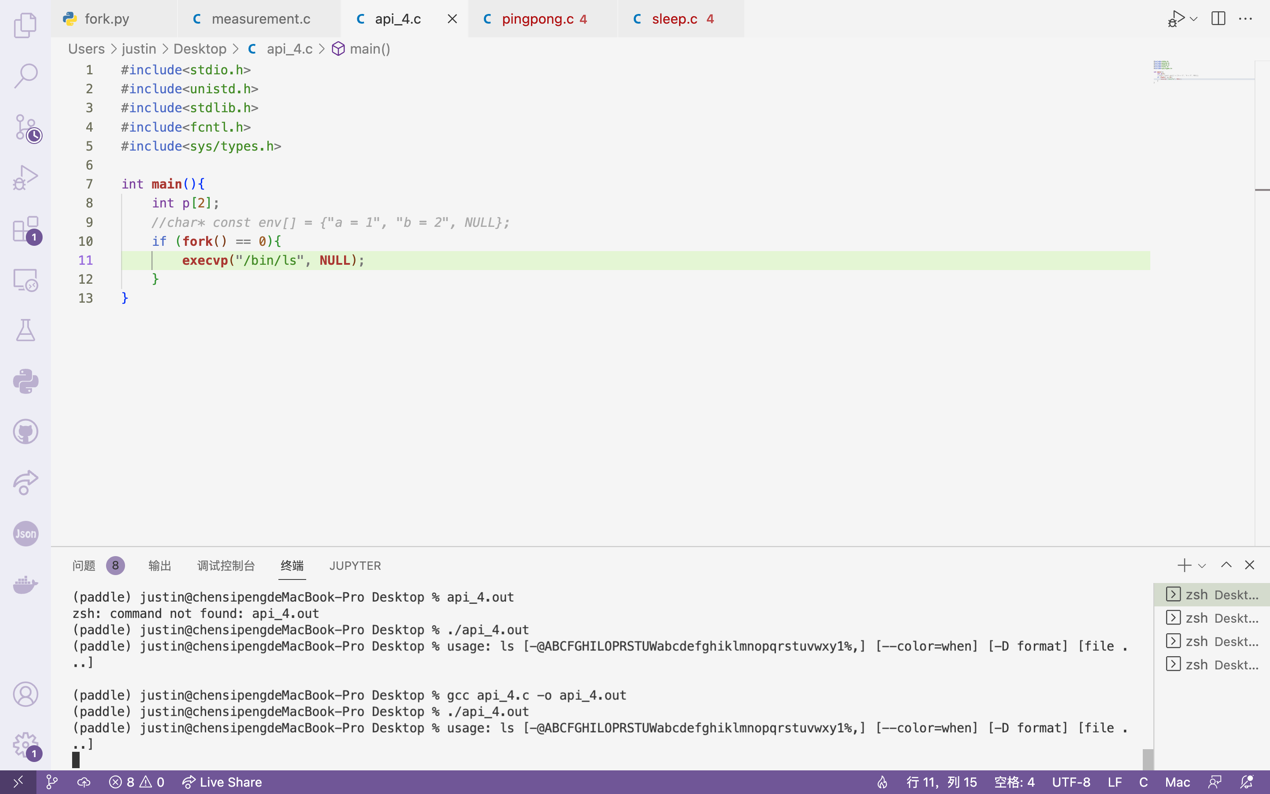
Execle() has a char\* const variable which enables us to pass the environment variable into the file that is to be executed.



Execlp() will automatically search the path according to the filename.

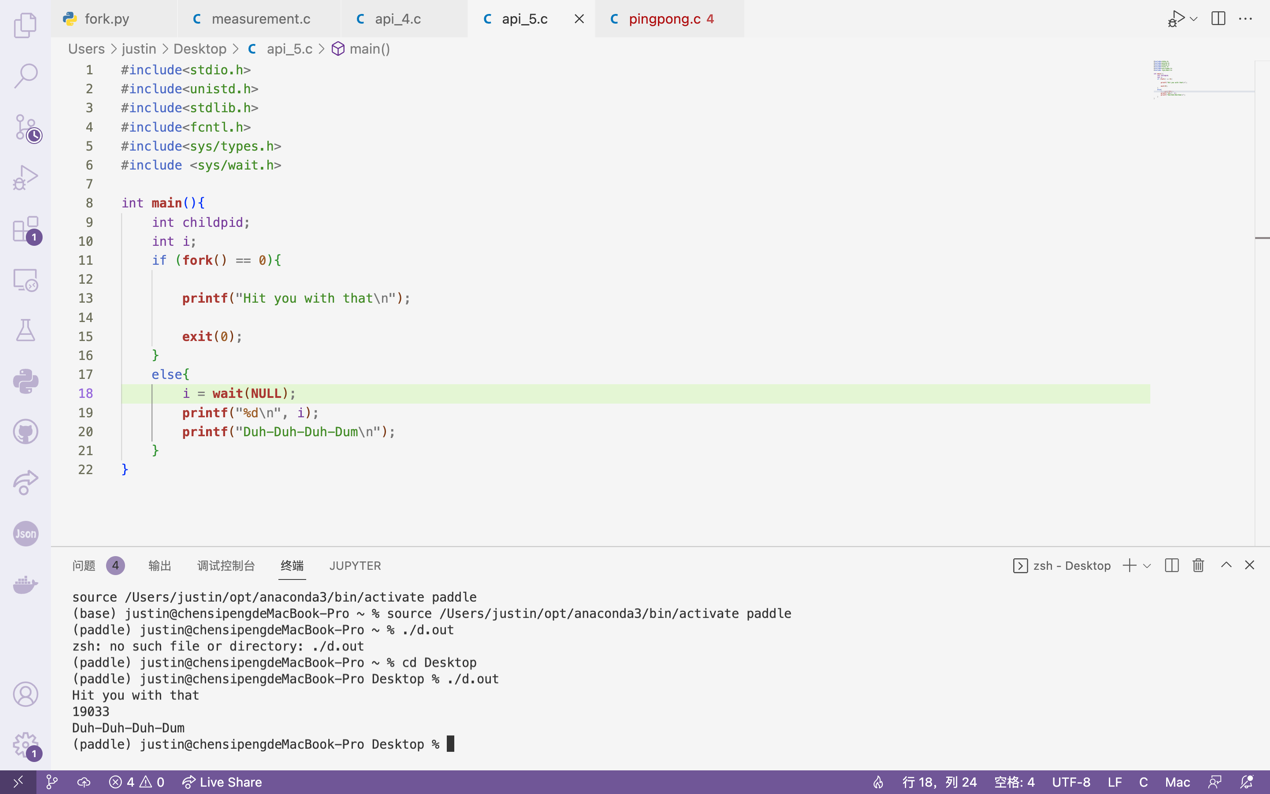


Execv() only allows one argument to be passed to the file.

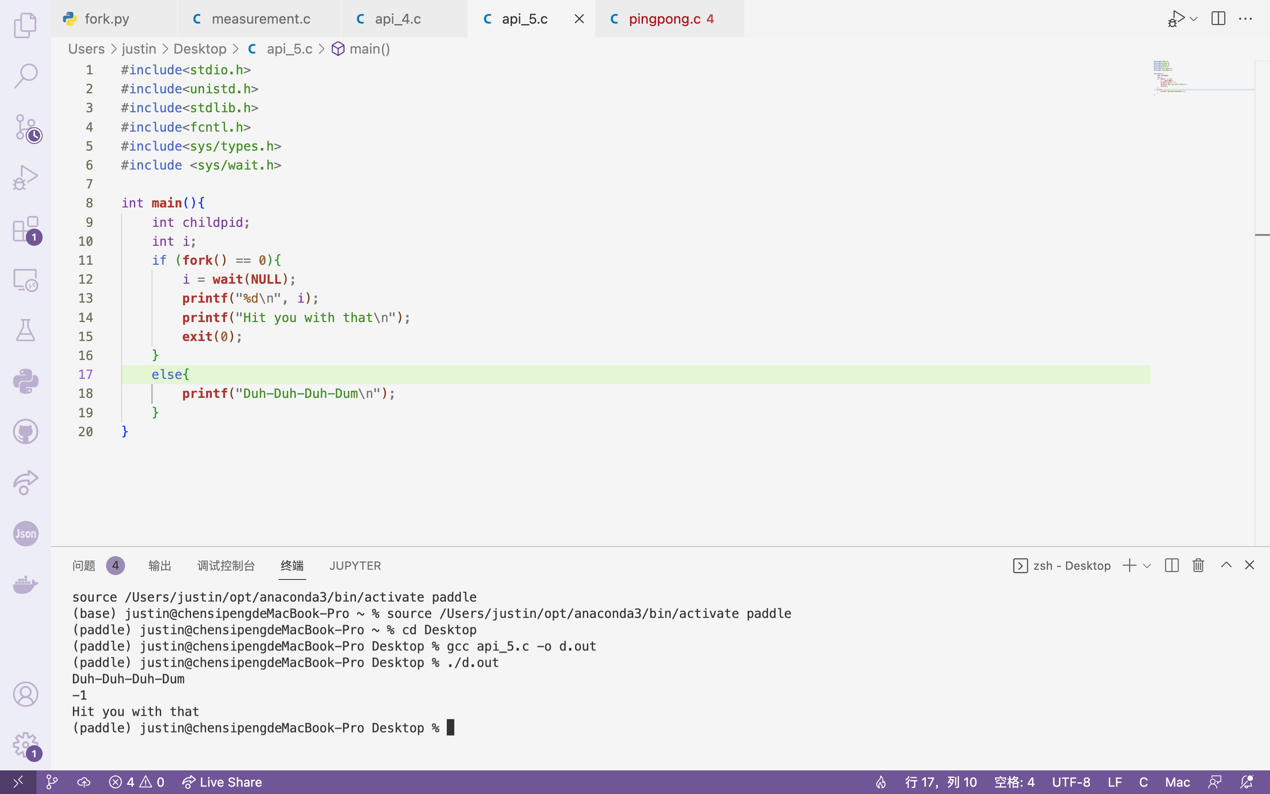


And finally, execvp() is a combination of execv() and the function to automatically search the path.

1. **Now write a program that uses wait() to wait for the child process to finish in the parent. What does wait() return? What happens if you use wait() in the child?**

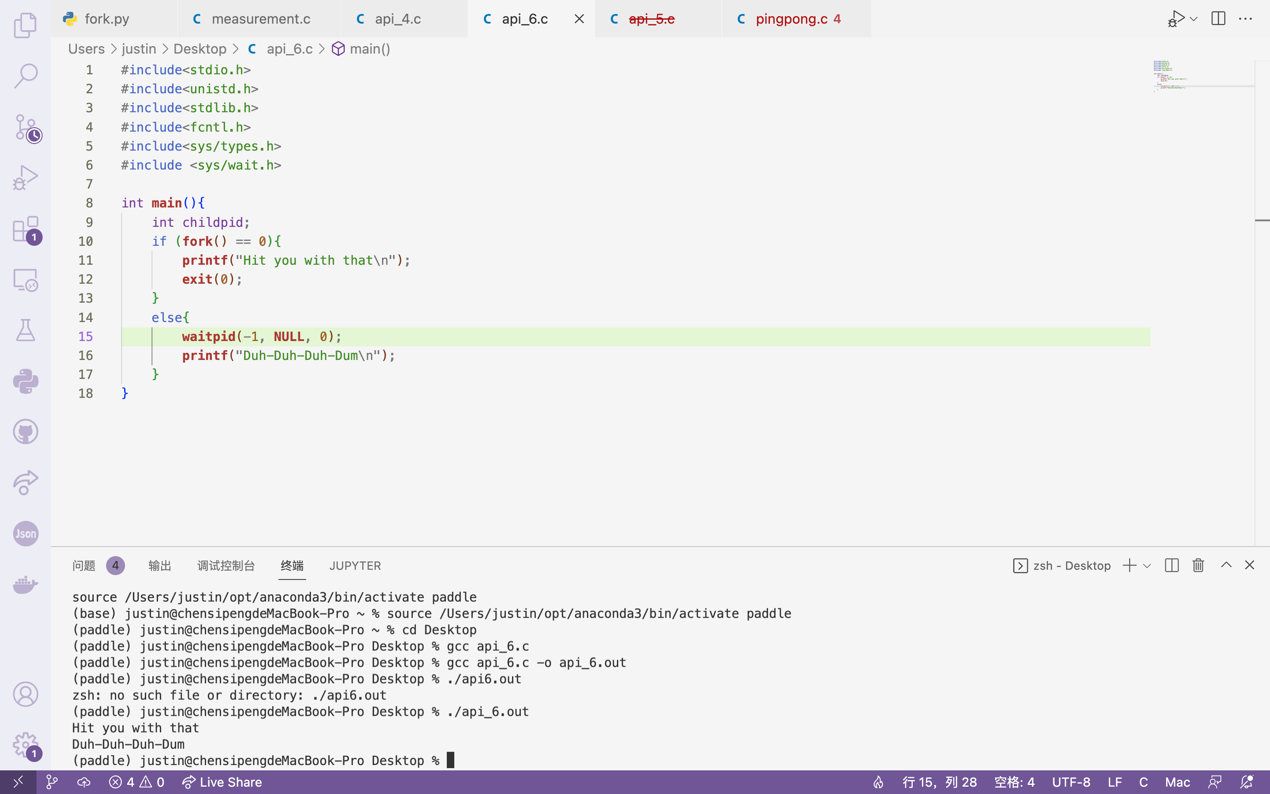


Function wait() guarantees the child process is always executed first and returns value 19033 in the case above, which is the PID of the child process that exits.



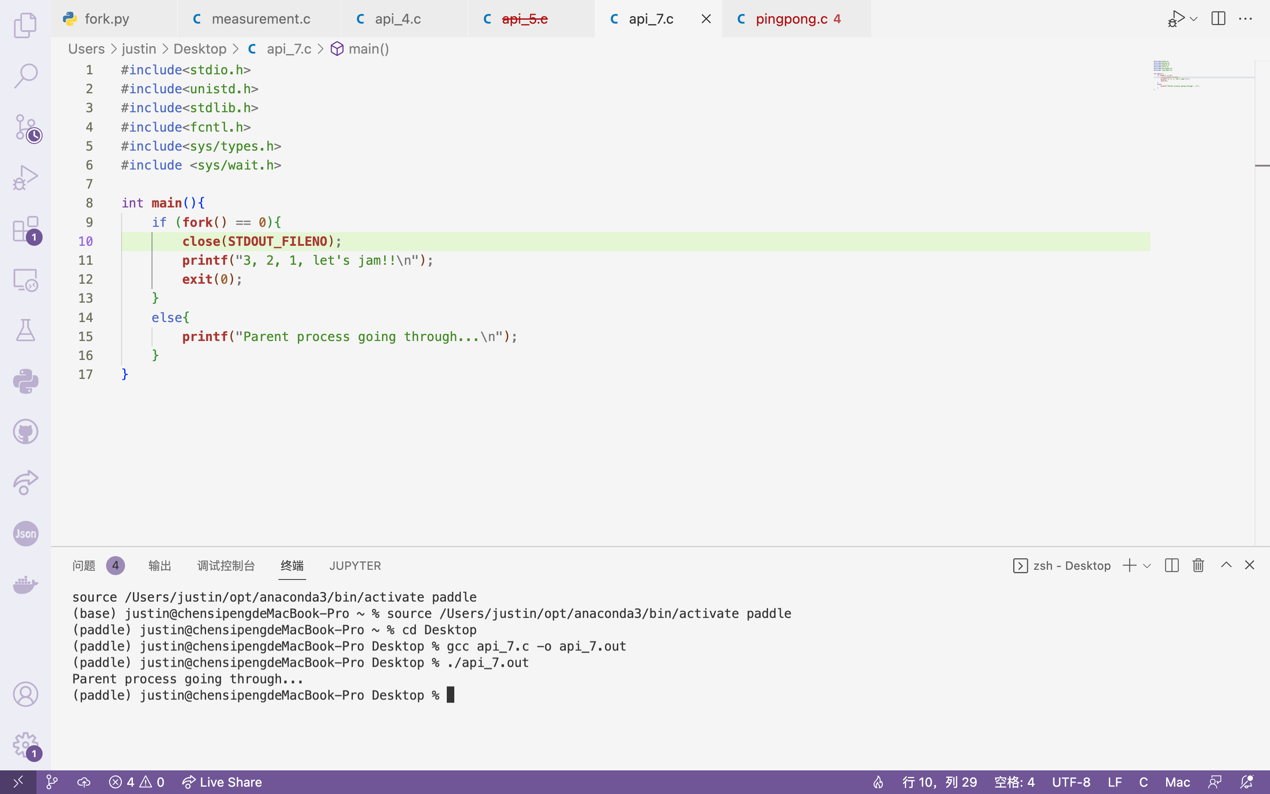
If the child process calls wait(), function wait() will return value -1, which is what happens when wait() is called in a process that has no child processes. Also, although the above case shows that when a child process calls wait(), its parent process will be executed first, this is actually a random outcome since wait() didn’t function at all as the child process that calls wait() has not a single child.

1. **Write a slight modification of the previous program, this time using waitpid() instead of wait(). When would waitpid() be useful?**



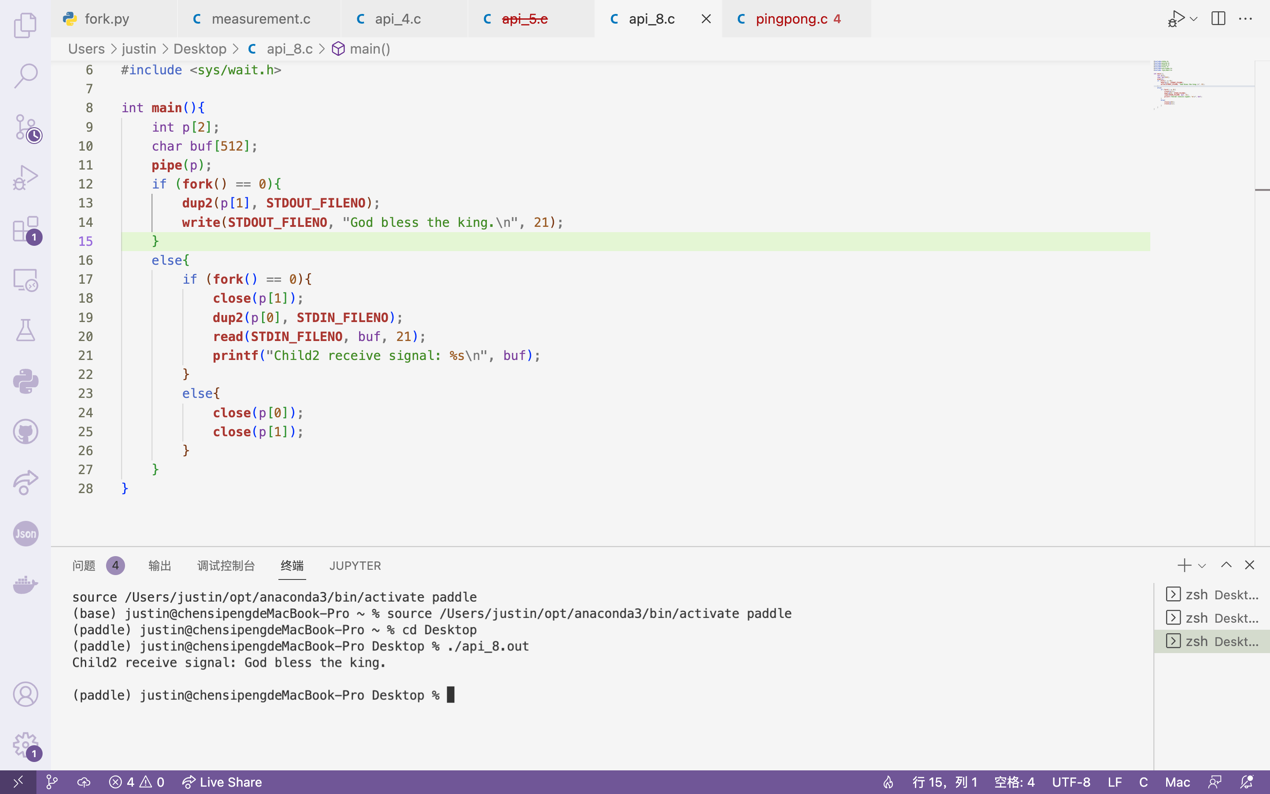
Waitpid() functions with the help of three parameters, namely pid\_t pid, int \*status, and int options. Pid tells the function which process it should wait for its exiting. \*status helps us diagnose why the child process exits. With the three parameters set as above, waitpid() functions just as a normal wait() will do. Because wait() will simply wait for any child process to exit, waitpid() would be of great help if we are to wait for a specific child to be executed.

1. **Write a program that creates a child process, and then in the child closes standard output (STDOUT FILENO). What happens if the child calls printf() to print some output after closing the descriptor?**



It can be seen from the screenshot above that after shutting down STDOUT\_FILENO in child process, “printf” will no longer print the corresponding content in the terminal. However, “printf” in the parent process will not be insusceptible.

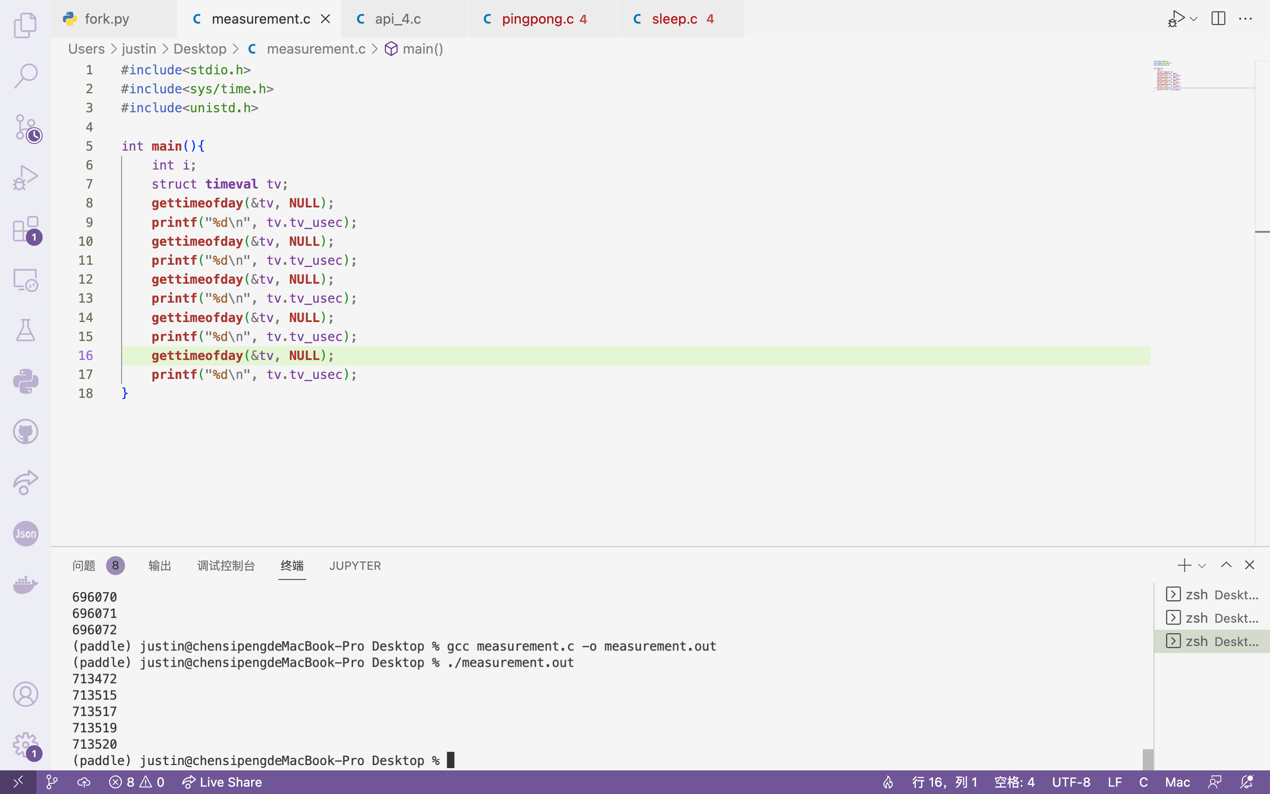
1. **Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the pipe() system call.**



Use dup2 to duplicate STDOUT\_FILENO in child 1 and STDIN\_FILENO in child 2. The screenshot above shows that child 2 receives the information in child 1’s standard output, which is “God bless the king.” and isn’t printed in the terminal as standard output has been redirected.

**Cpt 6**

**Homework (Measurement)**



The above shows the outcome of back-to-back calls of gettiemofday(), which indicates that though executing one single line of gettimeofday() will only take a few microseconds, the time consumed will still be quite random, ranging from 1 to almost 50 microseconds. Thus, in order to calculate the exact time spent on a system call, we are required to run a large number of iterations and get the average time consumed.

