

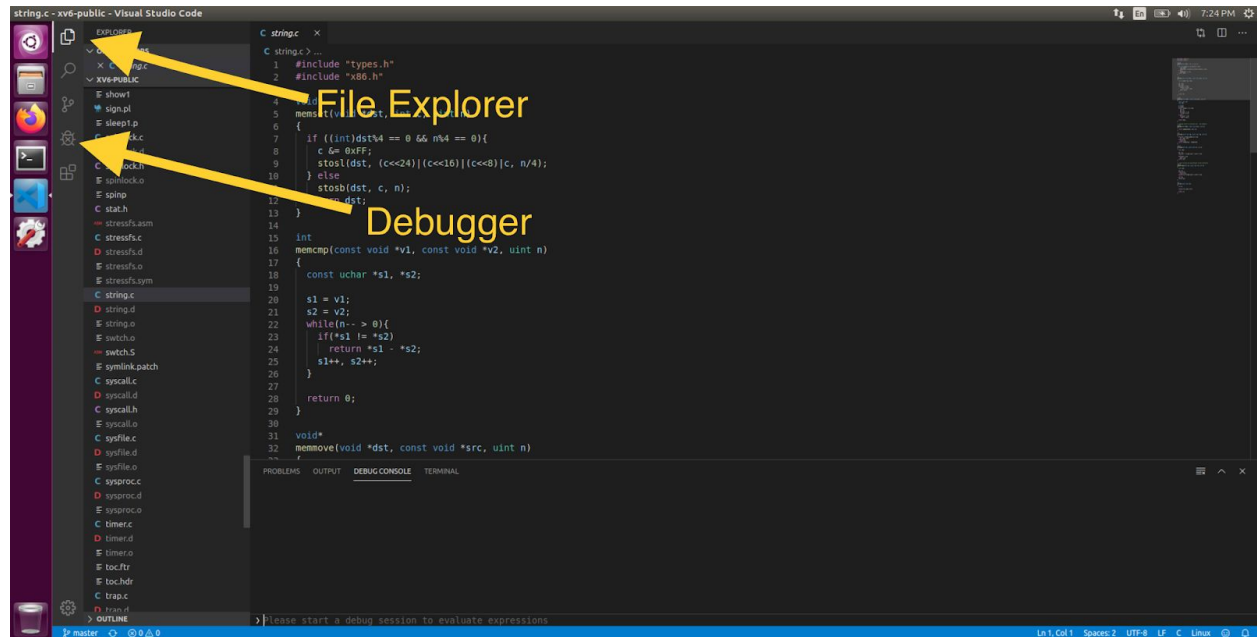
# How to Debug in XV6

Important distinctions:

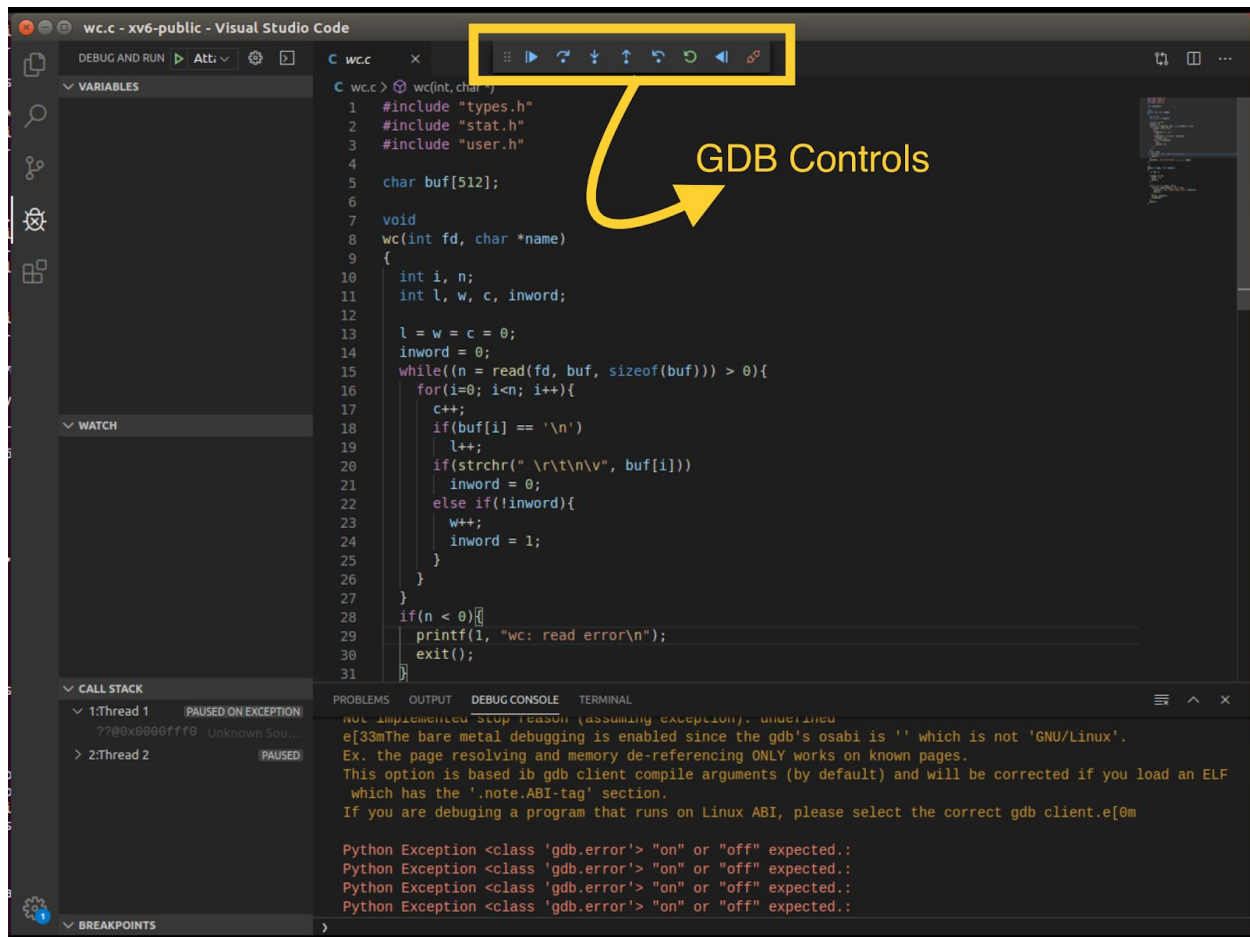
- There will be differences in debugging kernel land and user land programs. Your first step for debugging should be identifying which type of program you are trying to debug.
  - If it is a command line utility like `uniq`, then you are bound to be debugging user land.
  - If it is a syscall you are debugging, then you will be debugging kernel land.

## Using vscode

Vscode is a free IDE that allows for clean debugging of xv6.



When we are running the debugger, the gdb controls will be displayed up top here:



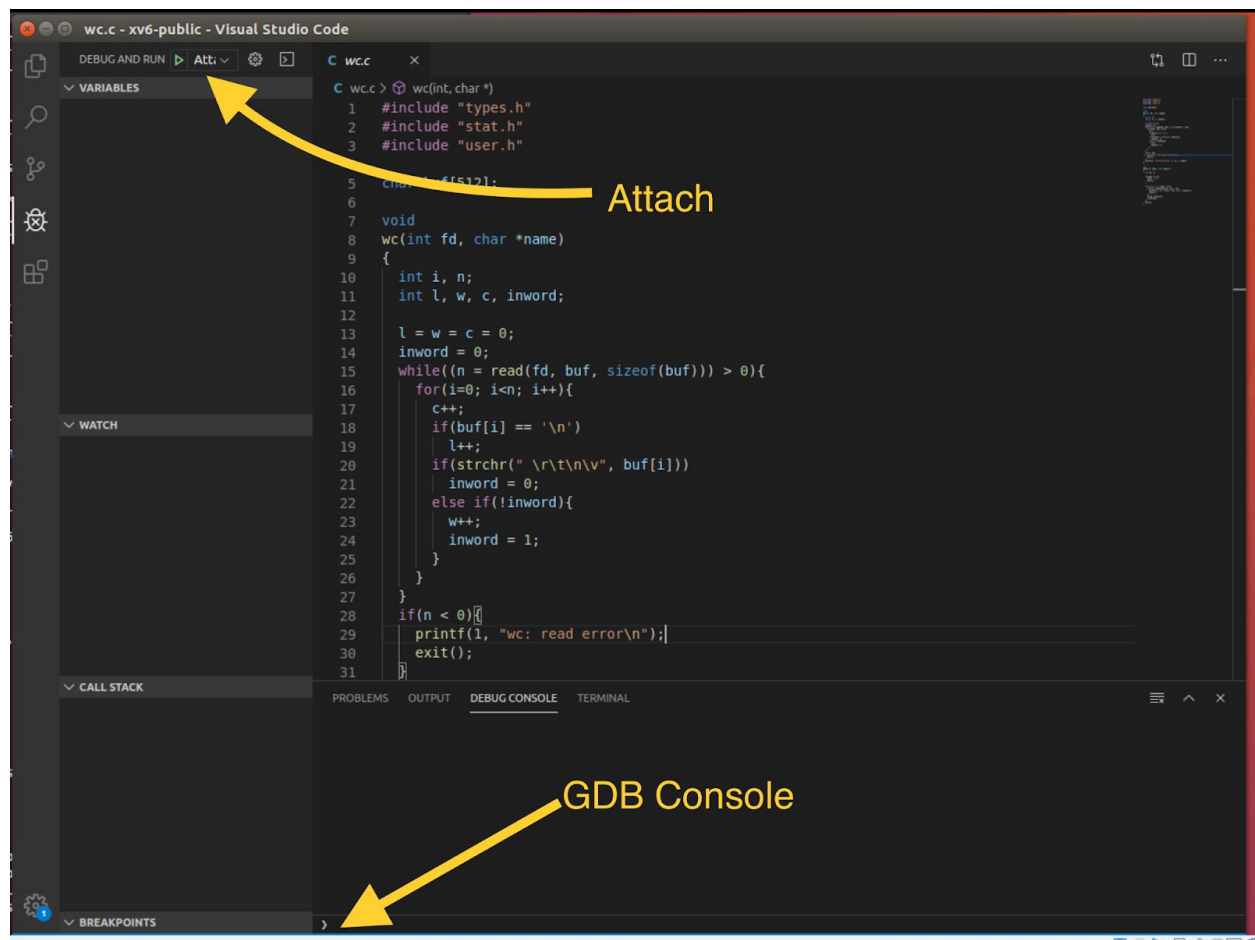
## Debugging user land programs

For this example, I will be showing how to debug the user land program wc. The first step would be to compile the program. Do this with `make xv6.img`. Once that is done, we can start the kernel with qemu in debug mode. To do this, run `make qemu-vscod`:

```
os3224@os3224 ~/xv6-public <master>
$ make qemu-vscod
mkdir -p .vscode
if [ -f .gdbinit ]; then rm .gdbinit; fi
sed "s/localhost:1234/localhost:26000/" < launch.json.tmpl > .vscode/launch.json
*** Now attach to qemu in the debug console ofb vscode.
qemu-system-i386 -serial mon:stdio -drive file=xv6.img,media=disk,index=0,format=raw -drive file=fs.i
mg,index=1,media=disk,format=raw -smp 2 -m 512 -display none -nographic -S -gdb tcp::26000
```

At this point, xv6 will be stopped and waiting for a vscode to connect to it. Navigate to the debug tab in vscode, and press the play button next to attach. You should see some output in the

debug console on vscode, and the terminal should display xv6 starting normally. You can now issue gdb commands to the debug console here:

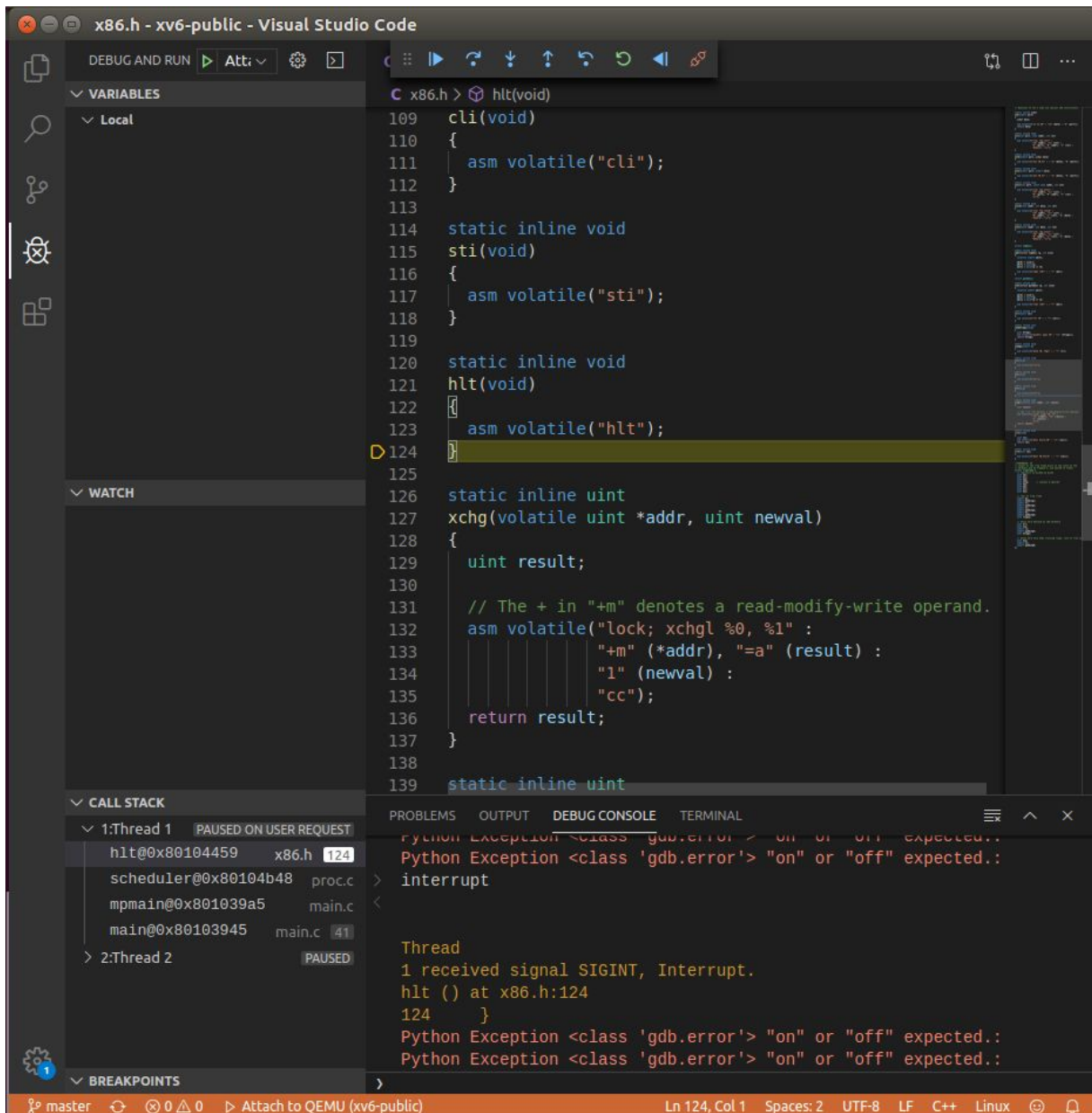


To control gdb, use these controls. They are standard debugging commands like stepping and continuing.

These next few steps need to happen in a specific order, or the debugging will not work. By default, when xv6 starts it's only looking to debug the kernel itself. We want to switch the program the debugger is looking at using the symbol-file command. Once this is the case, we can set our breakpoints in vscode. We can **NOT** set breakpoints without xv6 being stopped. If xv6 is still running, then any breakpoint we set will be ignored. We can stop the xv6 kernel with the interrupt command. So to recap, to set a breakpoint we need to (in this order):

- stop the kernel
- set the symbol file
- set our breakpoints

Stop the xv6 with interrupt (will stop where ever it was):



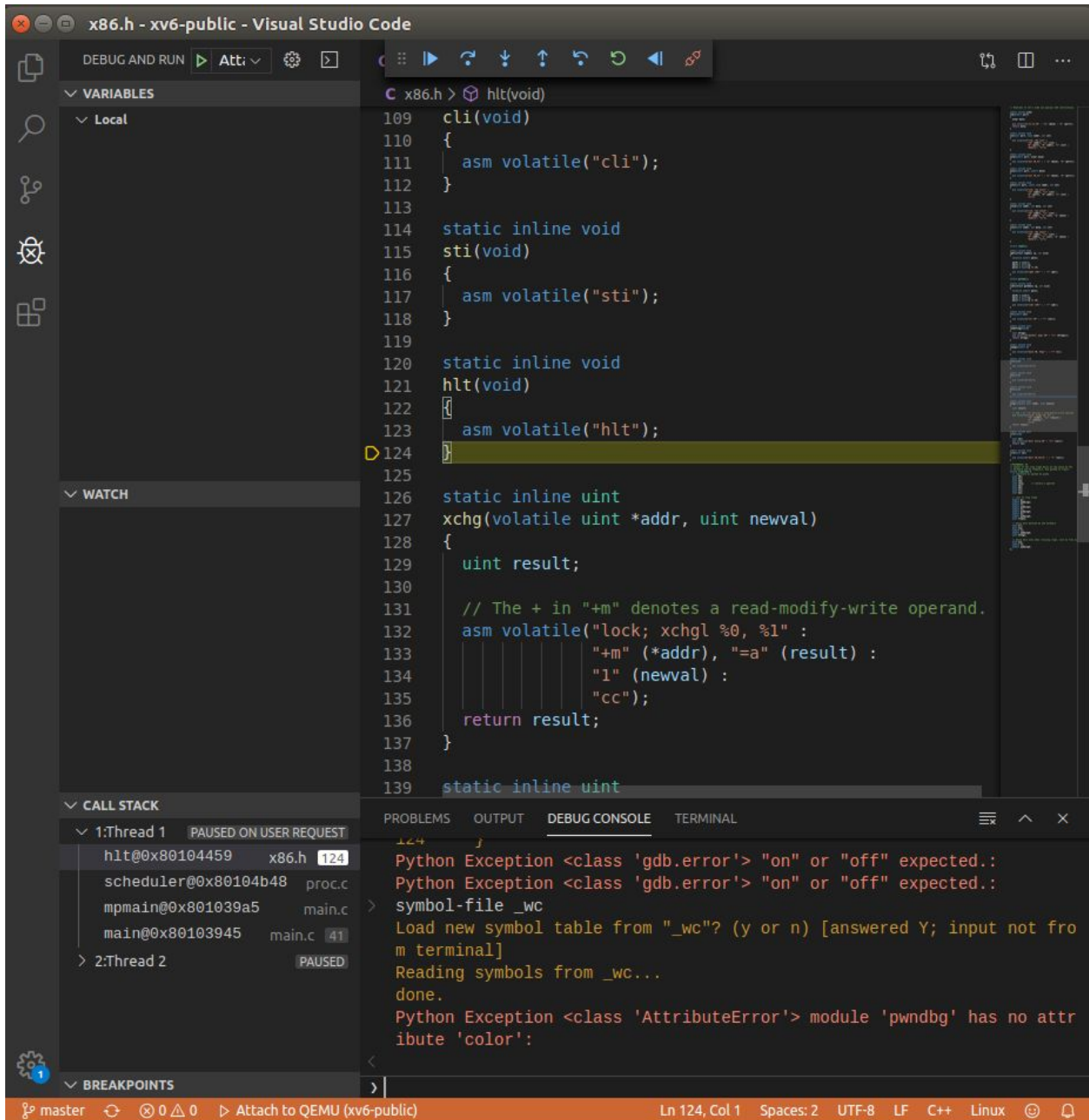
The screenshot shows the Visual Studio Code interface with the xv6 kernel source code open. The file `x86.h` is being edited, and a breakpoint is set at line 124, which is highlighted in yellow. The code includes functions for `cli`, `sti`, and `hlt`, along with a memory management function `xchg`. The left sidebar shows the `VARIABLES` and `WATCH` panels, both of which are empty. The `CALL STACK` panel shows the current execution path: `hlt@0x80104459` in `x86.h` at line 124, followed by `scheduler@0x80104b48` in `proc.c`, `mpmain@0x801039a5` in `main.c`, and `main@0x80103945` in `main.c` at line 41. The `DEBUG CONSOLE` panel shows the following output:

```
python Exception <class 'gdb.error'> "on" or "off" expected.:
Python Exception <class 'gdb.error'> "on" or "off" expected.:
interrupt

Thread
1 received signal SIGINT, Interrupt.
hlt () at x86.h:124
124  }
Python Exception <class 'gdb.error'> "on" or "off" expected.:
Python Exception <class 'gdb.error'> "on" or "off" expected.:
```

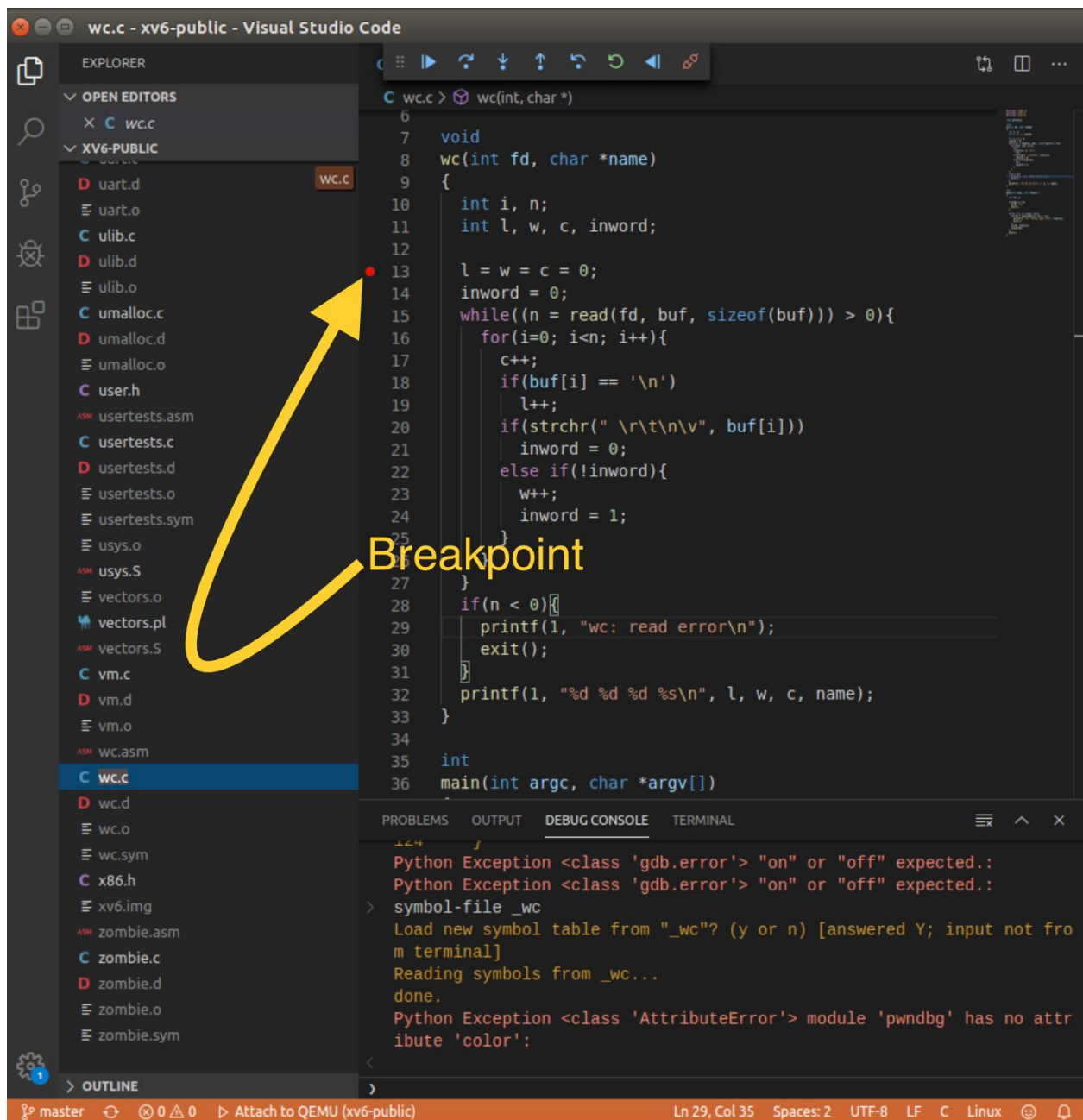
The status bar at the bottom indicates the current position is at line 124, column 1, and the file is `x86.h` in the `xv6-public` project.

Set the symbol file. This should be the name of the compiled program you want to debug. For userland programs like `wc.c`, it will be the underscore character followed by the name of the file. For example, `wc.c` will be compiled to `_wc`:

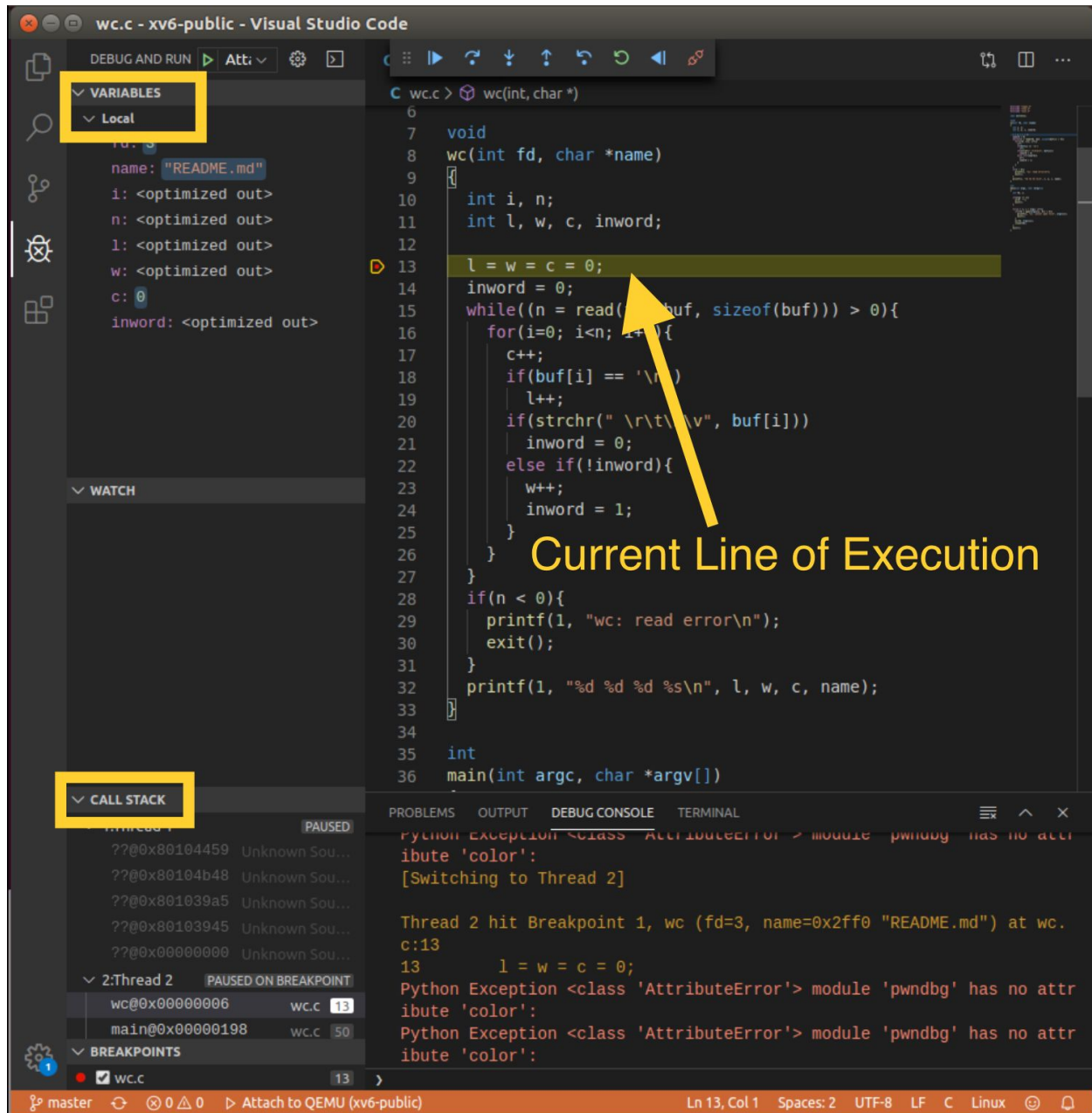


We can now set our breakpoints. To do this, navigate to the program you are debugging, then click just to the left of the line number you want to debug. If everything is successful, you should see a red dot next to the line you are debugging. Check the debug console to make sure you did not get any error when setting that breakpoint. If you did, you may not have done things in the correct order. All goes well, and you should see something like this:





Now that our breakpoints are set, and our symbol file loaded, we can resume xv6's execution and run the program we want to debug. You can either type `c` into the debug console (for continue) or click the continue icon in the gdb controls. The next step is actually running the program. If your breakpoint is hit, then the debugger will stop the program and show you stuff about it. You should see any local variable, along with the call stack and what line you are currently on. You can then control what the debugger does next with the gdb controls up top.

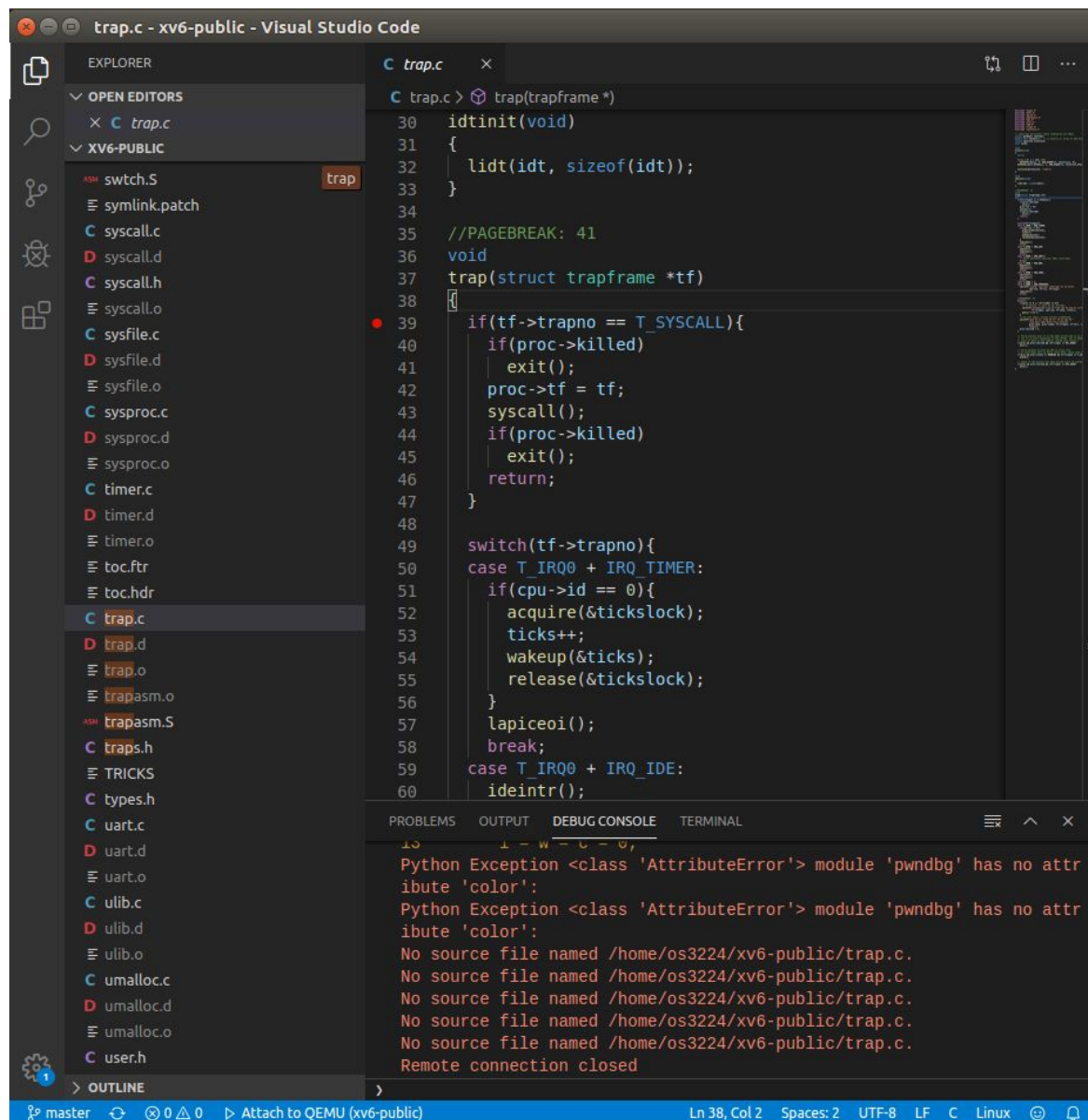


If you recompile the program, you will need to completely restart the debugger to get the new version of the program. Disconnect the debugger in vscode with the unplug icon in the gdb controls, then stop xv6 in the terminal. Repeat all the steps above to debug your changes.

## Debugging kernel land

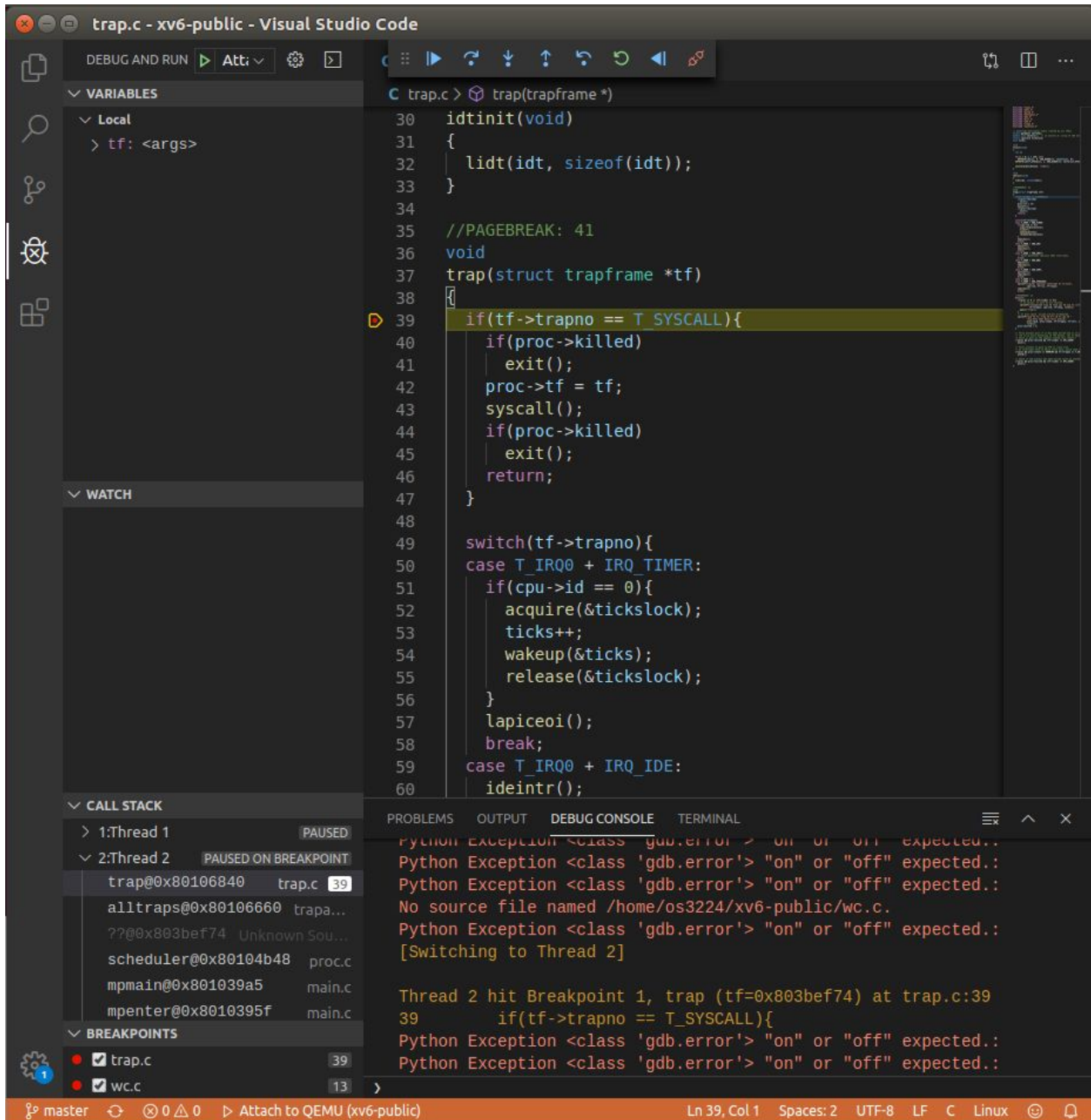
Debugging the kernel is much more simple than user land programs. When you start xv6 with the make qemu-vscode, it is already set up to debug the kernel, so we need not switch any controls there. We'll be skipping some steps that are already described above.

Say we want to debug the trap function in trap.c. To debug this, all we need to do is set the breakpoint in vscode by clicking to the left of the line we want to stop at:



We can then start xv6 in debug mode by running `make qemu-vsc` in the terminal. Once we then click attach in the vscode debugger, we should be met with something that looks like this in vscode:





Controlling the debugger is the same in user land. Just use the gdb controls on top to step, continue or disconnect.

If you make changes and want to debug them. You will need to recompile, then restart this whole process to debug the updated xv6. That means disconnecting the vscode debugger, restarting xv6 in the terminal, then re-attaching the vscode debugger to the new xv6.