xv6 Debug

写到ppt里有点慢，先直接贴成pdf了

1. 用GDB直接调

用GDB把一般的大象放进冰箱需要三步：

1. `gcc example.c -g -o example` 编译成带有调试信息的可执行文件；
2. `gdb example` 用 `gdb` 运行要调试的程序 `example`；
3. 使用*gdb commands*来控制程序。

用GDB把内核的大象放进冰箱需要五步：

1. `make qemu-nox-gdb` 或者 `make qemu-gdb`；
2. 在另一个终端（shell，窗口，session，whatever）打开 `gdb`；
3. 首先用 `file` 读取要调试的文件，例如如果要调内核本身，就需要 `file your_xv6_dir/kernel/kernel`；
4. 然后用 `target remote localhost:26000` 来连接(端口号默认是26000，可以自己在makefile里调)。
5. 最后跟调试普通程序一样调试内核。

用GDB把xv6里其它程序的大象放进冰箱，和内核的区别是 `file` 的程序是你要调的程序，打断点也应该在你要调的代码里打。然后在xv6里执行你要调的程序，就会自动在断点处停止的。

> .gdbinit文件可以帮助配置，xv6提供了配置好的.gdbinit，在xv6/tools/目录下（dot-gdbinit），需要使用的话改名为.gdbinit然后放在你要放的地方就行。
>
> gdbinit(5) - Linux manual page (man7.org)

2. 用VSC调

本质上还是用gdb，只不过vsc帮你把这些操作全都可视化了，方便快捷。

首先，你要能正确的打开vscode。

请用vscode打开一整个文件夹（aka 一个工作区），而不是用vscode单独打开某个文件。

例如：

```
subwoy@DESKTOP-ENVI0R2:~/OSLab/project3b-s7bw0y/xv6$ ls
FILES  Makefile  README  bootother  fs  fs.img  include  initcode  kernel  log.txt  tools  user  version  xv6.img
subwoy@DESKTOP-ENVI0R2:~/OSLab/project3b-s7bw0y/xv6$ code .
```

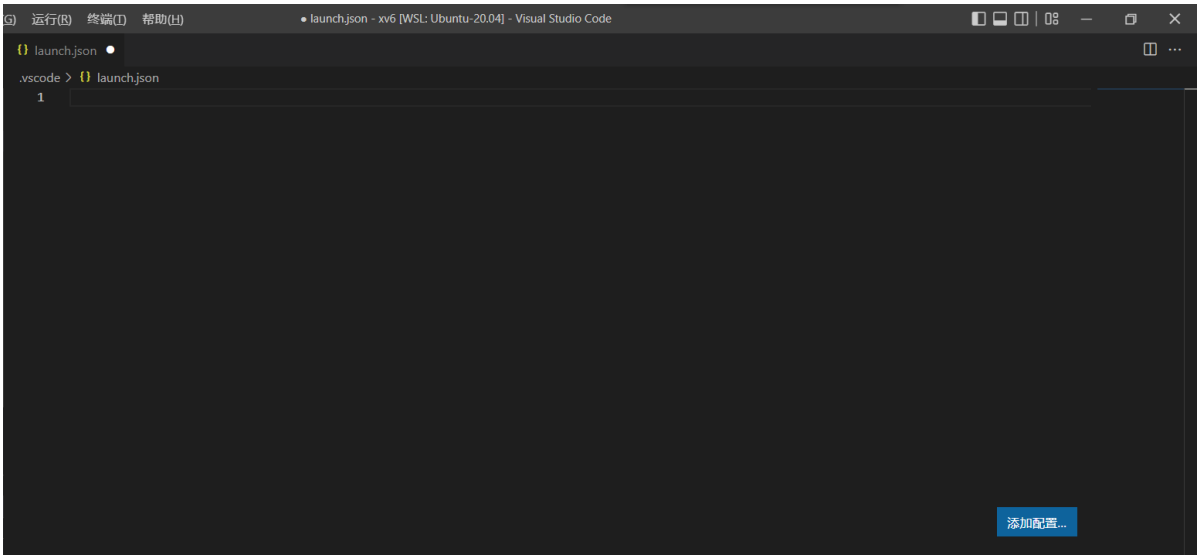
表明用vsc打开当前目录。这样才能发挥vsc在工作区的作用。

如果你没有在一个工作区里配置过vscode，那你需要手动在工作区目录下创建一个名字是 `.vscode` 的文件夹，用来存放配置。

因为我们是调C程序，所以你需要在拓展里安装相关插件。（`C/C++ Extension Pack` 就足够了，**不要安装和同时启用过多拓展插件。**）

一般情况下你在vsc设置的工作区配置都会自动以json文件（什么是json？非常简单，五分钟就能理解。可以参见JSON官网,或者JSON Introduction (w3schools.com)）的形式放在这里，所以你可能会看到其它文件，例如 `settings.json, tasks.json, launch.json` 等。

我们~~原神~~启动xv6需要的是launch.json文件，它告诉了vsc该如何运行调试。如果`.vscode`里没有，你需要手动创建一个。



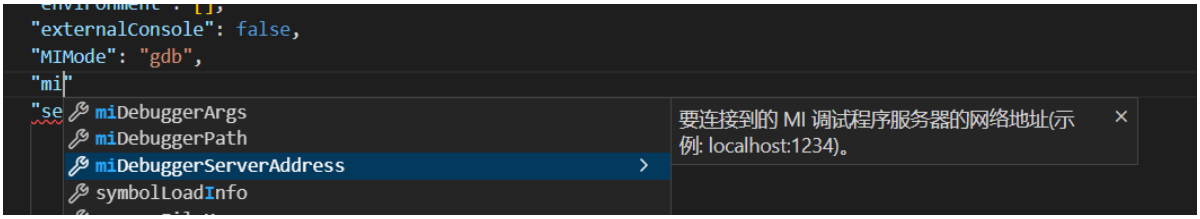
vsc识别之后会在右下角弹出一个“添加配置”按钮。相当于一个向导，点击之后会给你一些模板。

这里可以直接用第一个模板，然后修改参数就能直接用了。

```jsonc
{
    "configurations": [
    {
        "name": "(gdb) 启动",
        "type": "cppdbg",
        "request": "launch",
        "program": "输入程序名称，例如 ${workspaceFolder}/a.out",
        "args": [],
        "stopAtEntry": false,
        "cwd": "${fileDirname}",
        "environment": [],
        "externalConsole": false,
        "MIMode": "gdb",
        "setupCommands": [
            {
                "description": "为 gdb 启用整齐打印",
                "text": "-enable-pretty-printing",
                "ignoreFailures": true
            },
            {
                "description": "将反汇编风格设置为 Intel",
                "text": "-gdb-set disassembly-flavor intel",
                "ignoreFailures": true
            }
        ]
    }
    ]
}
```

所有参数都是字面意思，如果不理解参数意义可以把鼠标悬停在上面，会有提示。如果想添加其它配置，也会有补全提示和解释。

由于我们是远程调试，所以需要一些额外的配置，例如：

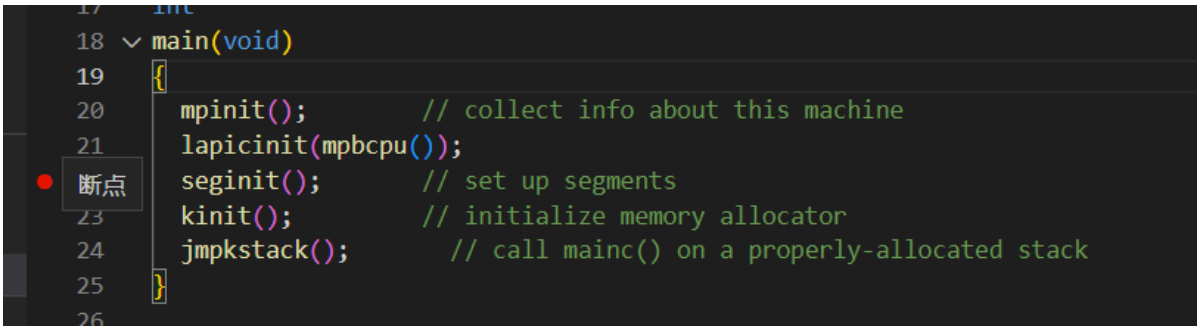


直接配成 `localhost:26000` 就可以了。

然后把 `"program"` 这一项改成你想要调的程序。

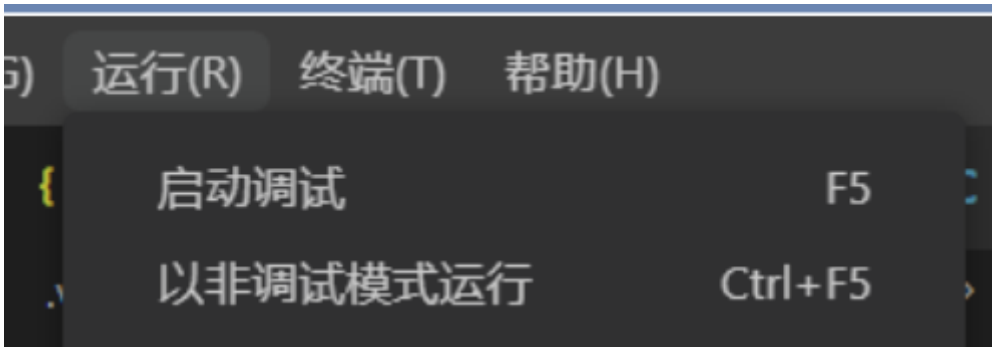
配置完成之后记得保存。

然后就可以在代码里打断点了，比如main里面随便一行：

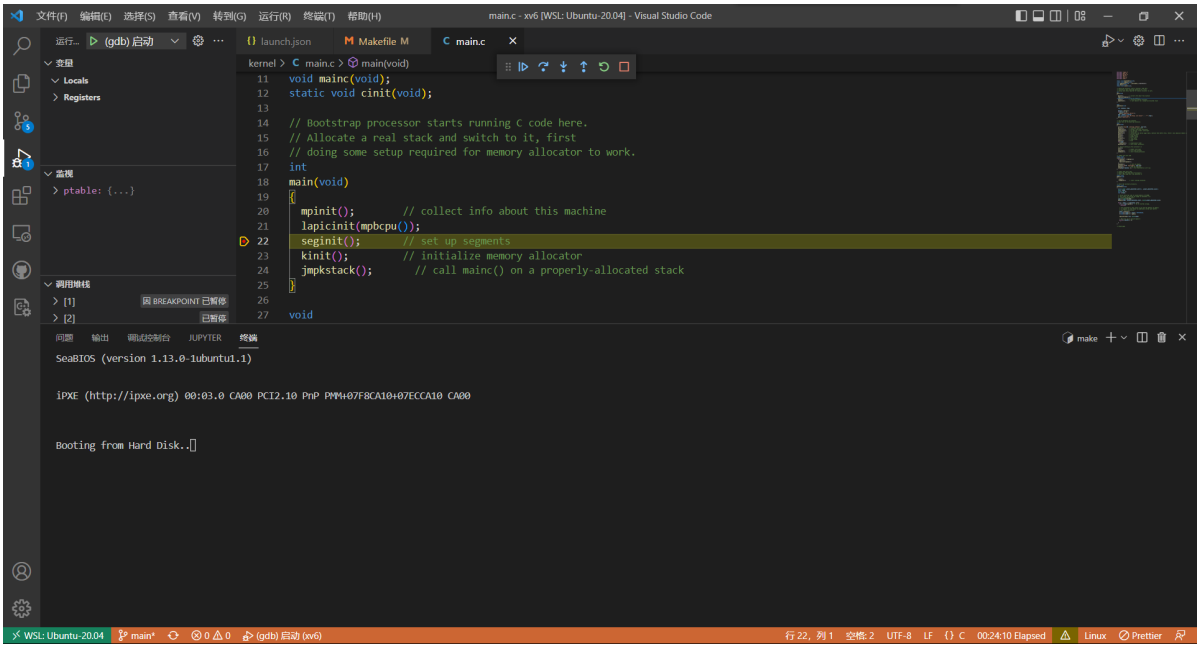


点一下就行。

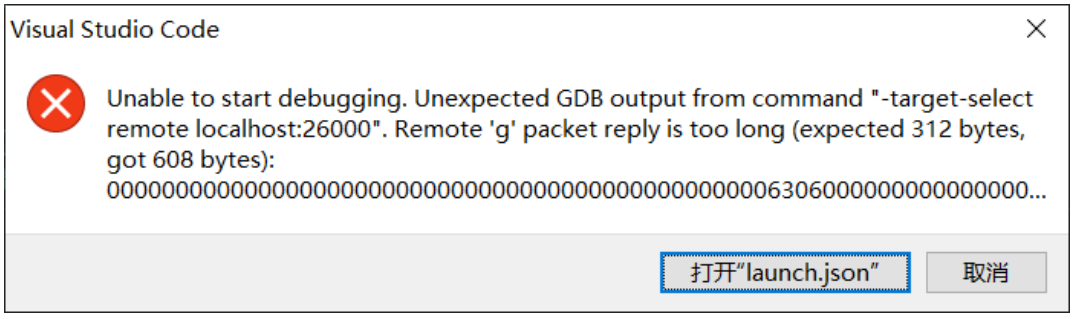
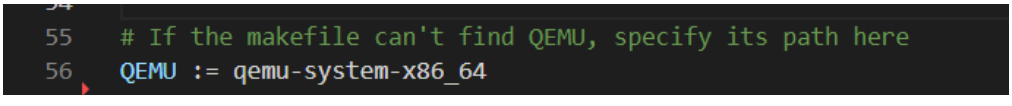
在`make qemu-nox-gdb`或者`make qemu-gdb`之后在vsc里直接按F5启动调试（快捷键）。



没有错误的话此时应该已经正常开始运行并且调试了。



所有工具都是可用的，包括左侧的变量和监视栏，悬浮的调试栏。

> Tips:
>
> 1. stopAtEntry建议置true，相当于自动在main开始时断点。
> 2. 提示找不到gdb？可能需要在配置里加上`"miDebuggerPath": "/bin/gdb"`。
> 3. 报一个神奇的错？
>
> 
>
>
> 这是因为我们的Makefile里默认调用的是`qemu-system-x86_64`，而实际用的架构是i386。所以有两个解决方案：
>
> 1. 安装qemu-system-i386，然后把Makefile里的
>
> 
>
>
> 改成`qemu-system-i386`

2. 或者可以在`launch.json`里的`"setupCommands"`列表中添加一项：

```
{
    "description": "设置架构",
    "text": "-gdb-set architecture i386:x86-64:intel",
    "ignoreFailures": false
}
```

3. 内核的部分代码涉及到汇编，同时也有一些系统中断，所以某些情况下单步跳过和单步调试会失败。较为稳妥的办法是在需要的地方打断点，然后使用“继续”来运行到断点处。
4. Q:怎么断点和我调的代码不在同一行？怎么光标乱飞？

   A:重新编译。`make clean && make qemu-nox-gdb`

我的配置（参考）

```
{
    "configurations": [
    {
        "name": "(gdb) 启动",
        "type": "cppdbg",
        "request": "launch",
        "program": "${workspaceFolder}/kernel/kernel",
        "args": [],
        "stopAtEntry": true,
        "cwd": "${fileDirname}",
        "environment": [],
        "externalConsole": false,
        "miDebuggerPath": "/bin/gdb",
        "miDebuggerServerAddress": "localhost:26000",
        "setupCommands": [
            {
                "description": "为 gdb 启用整齐打印",
                "text": "-enable-pretty-printing",
                "ignoreFailures": true
            },
            {
                "description":  "将反汇编风格设置为 Intel",
                "text": "-gdb-set disassembly-flavor intel",
                "ignoreFailures": true
            },
            // {
            //     "description": "设置架构",
            //     "text": "-gdb-set architecture i386:x86-64:intel",
            //     "ignoreFailures": false
            // }
            // {
            //     "description": "设置fork",
            //     "text": "-gdb-set follow-fork-mode child",
            //     "ignoreFailures": false
            // }
        ],
    }
    ]
```

```
}
```

3. xv6自带的调试方法

首先，万能的`printf`调试法。xv6内核里提供了`cprintf`用于输出。

> 有些时候`cprintf`会失效，因为某些原因。

其次，`proc.c`文件中提供了一个方法`void procdump(void)`。

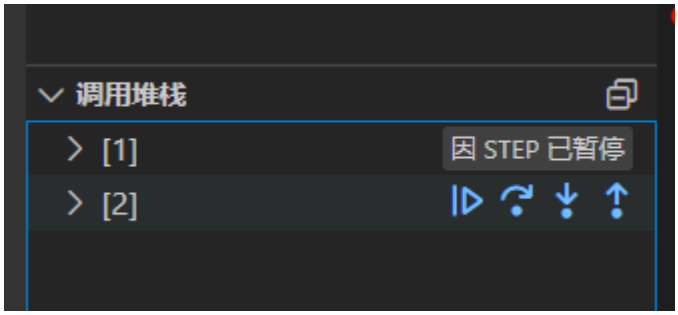
```
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void
procdump(void)
...
```

> 所以在xv6中按组合键`Ctrl+p`就可以自动调用一次该方法。
>
> p.s. 在vsc的终端里敲这个组合键可能会与vsc本身的热键冲突，所以建议在另一个单独的终端里运行xv6

最后，xv6还提供了一个用户程序`usertests`用于系统自检。一般来说想要通过测试至少要能通过系统自检。

进入xv6之后运行`./usertests`就会自动开始自检了。你也可以自行修改里面的部分，来执行自己的测试。

> 每次重新编译之前，`usertests`都只能执行一次。再次执行会提示已经执行过一次。所以你需要重新编译运行一遍xv6。

> ***One More Thing***: 请注意xv6的Makefile默认使用了两个CPU：
>
> ```
> # number of CPUs to emulate in QEMU
> ifndef CPUS
> CPUS := 2
> endif
> ```
>
> 所以你可以在左侧工具栏里看到调用堆栈里有两个线程：
>
> 
>
>
> 这就代表了两个CPU。
>
> Hint: 某些test中，会把CPU数改成1。