理论部分：

1.%rbx；%rbp；%r12；%r13；%r14；%r15

2.naive\_func:

endbr64

movq (%rsp), %rax

movq %rax,(%rdi)

xorq %rax,%rax

ret

3.传递的参数只有一个，不需要开栈空间存参数；

没有使用callee-saved寄存器，不需要临时存储其值；

没有调用别的函数，操作不需要用到栈空间

所以不需要开辟新的栈空间，所以不用对rsp，rbp进行相应的操作

4.栈指针%rsp和栈底指针%rbp

5.c++：

需要包含<exception>头文件

以及using namespa std；

格式：

try{

一些操作

if（某条件）

throw domain\_error（“错误名称”）

} catch（domain\_error e）{

一些操作

}

domain\_error可替换成其它类型的error

含义：

首先执行try后面的代码块。如果某处抛出了一个错误，就跳转到错误类型对应catch中，执行catch中的代码块。catch可以有多个。

6.在执行try中语句前保存“上下文”，正常情况下要求保存上下文的函数返回0。如果返回值是0就执行try的部分。如果返回值不是0（代表是跳转过来的）就执行catch的部分。

抛出异常的throw就恢复try中保存的上下文，并且把记录操作的返回值置为非0。

int e=record()

if(e){

do try;

if(condition)

recover(errortype);

}

else if(e== errortype 1){

do catch1;

}

else if(e== errortype 2){

do catch2;

}

7. 0

1

2

3

4

5

6

7

8

9

8.调用generator时，保存当前“上下文”。

当遇到yield时，保存当前“上下文”，再恢复到调用generator的上下文。

9.只在try-catch的语句里调用generator。

实验：

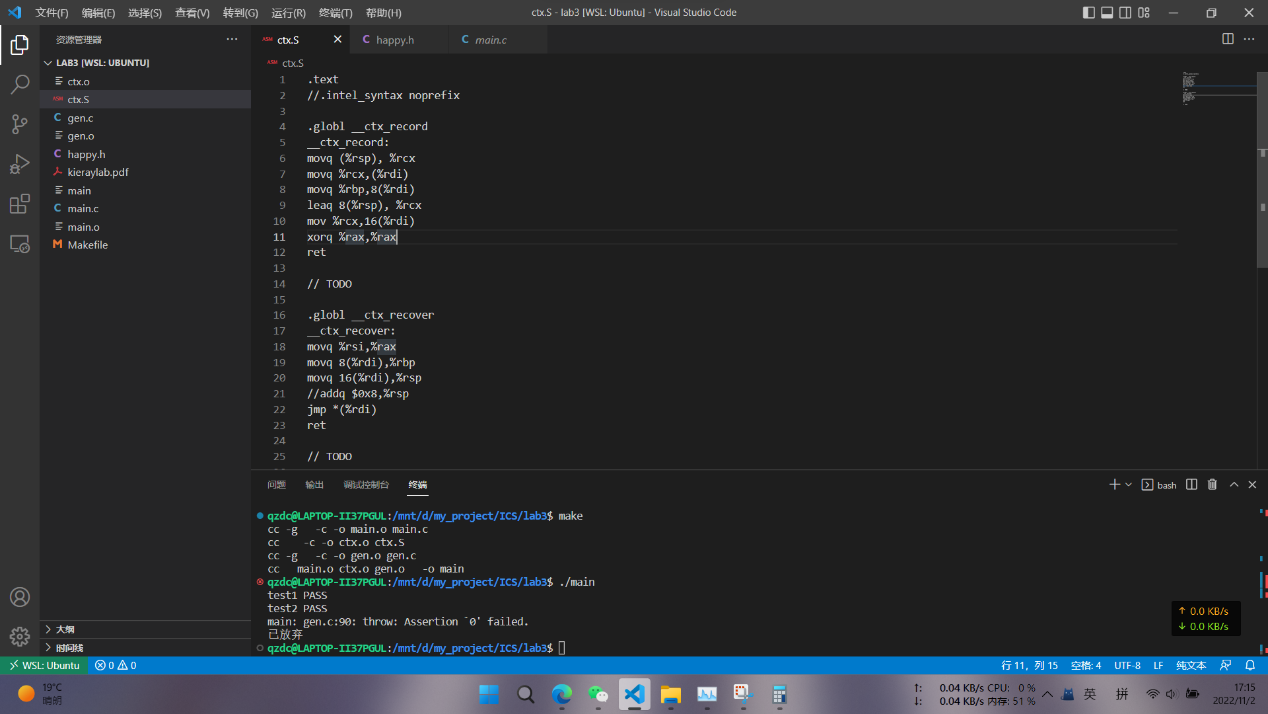
test1 2：

由于record函数传递的参数只有一个，不需要开栈空间存参数；没有使用callee-saved寄存器，不需要临时存储其值；没有调用别的函数，操作不需要用到栈空间。

所以record函数不需要开辟新的栈空间，所以只是rsp-8，存放了record的返回地址。因此，需要保存的有rsp指向的内容，即返回地址，还有rsp+8的值，即调用record前的栈顶，还有rbp。

recover函数首先将返回值设为第二个参数，然后恢复rbp与rsp，无条件跳转到记录的返回地址的位置。

此时\_\_ctx\_type为一个128字节的字符数组。成功通过test1 2



test3 4

由于需要进栈、出栈的操作，通过链表实现，就需要把\_\_ctx\_type改为结构体，内容是128个字符和一个\_\_ctx\_type类型的指针，存放下一个ctx。

push函数是先新建一个节点指针，然后让其指向传入的\_\_ctx\_type，将其链接到原有的\_\_eh\_list上，然后更新\_\_eh\_list为新的节点指针。

pop函数需要将原有的\_eh\_list转为\_\_ctx\_type指针类型，然后取出首元素，将\_\_en\_list更新其下一个\_\_ctx\_type类的指针，最后返回取出的首元素。

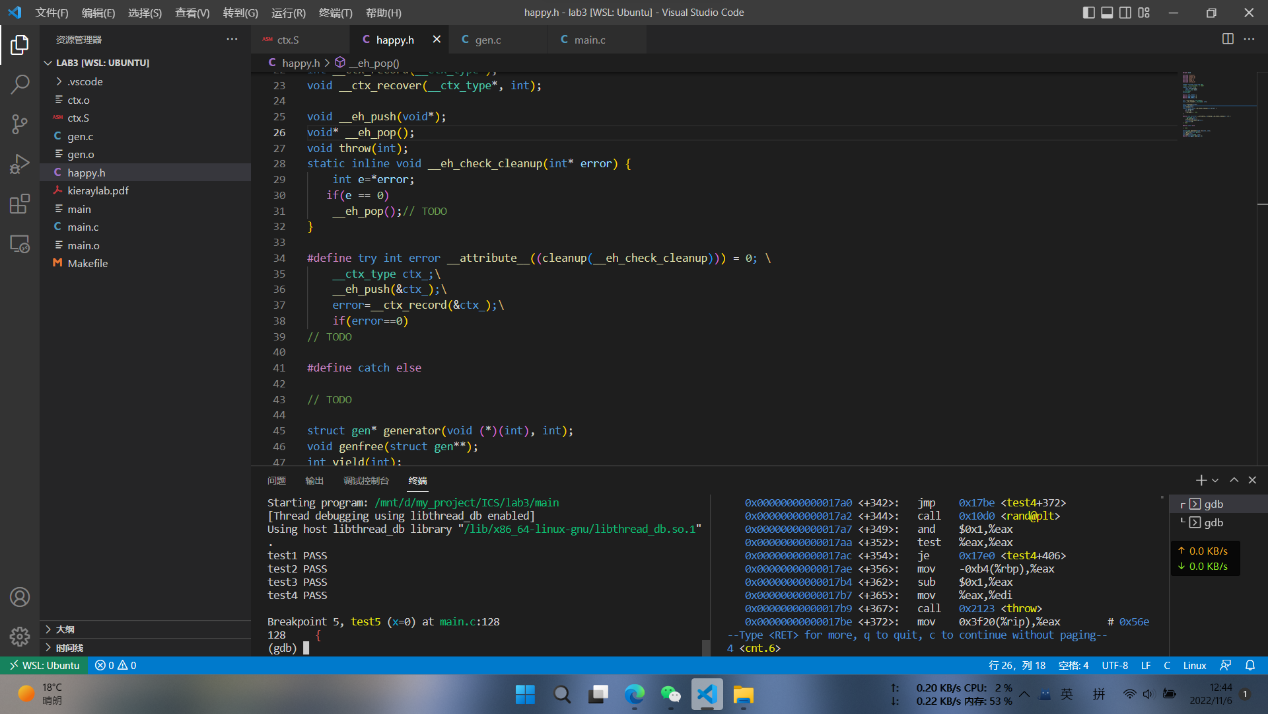
如理论部分所设想的，将try宏定义为：先保存上下文，就是创建一个\_\_ctx\_type类型的变量，然后ctx进栈。然后调用record函数，并且将error置为其返回值。如果error=0，代表是自动第一次执行到这，没有异常，就运行try后的代码块。如果error！=0，就说明这是恢复到这的，存在异常，应该运行catch后的代码块。

相应的，catch就是else。

对于throw函数，将\_\_eh\_list首元素取出，然后pop掉，然后恢复到刚刚取出的首元素的上下文，传入error。

最后，如果是正常结束，即不需要throw，所以先前入栈的ctx应该被pop出。如果正常结束，error应该是0。所以在cleanup中，检查error是否为0，是就pop出栈。

由于test5会持续输出e，导致看不到test4pass，需要gdb后断点才可以看到。



test5 6

对于yield函数，需要先把本generator的调用者的data置为value，达到传递“返回值”的功能。然后保存当前generator的上下文，将其返回值放在e里。然后根据e是否为0，即是否是正常的record返回而不是recover返回的，如果是正常返回，就需要切换到其调用者的上下文，以达到参数传递的效果。如果不是正常返回，即e非0，就说明是当前generator调用的generator的返回，就返回value，以保证和开始的send中的参数一致。

对于send函数，由于要跳到目标generator的上下文，所以需要让目标被当前generator调用，所以修改gen的调用者。然后保存上下文，记录返回值。如果正常返回，即是record返回的，就代表需要恢复上下文到目标generator里，同时由于发生跳转，所以更新当前generator为目标generator。如果不是正常返回，就代表是由其它地方恢复过来的，就需要继续返回，把当前gen更新为当前gen的调用者。

由于最开始的recover函数是通过修改rsp，用jmp指令伪造的返回，导致recover函数永远不会正常返回。由于这个栈空间是自己开辟的，所以需要recover配合，就不能伪造返回，需要让recover正常返回，只是恢复到record里栈的情况。同时，record也不能记录rsp+8，直接记录rsp即可。

除此之外，为了保证执行f（arg），需要把rdi设置为arg，就是record里增加的参数，地址是rdi+24。因为只有在配置generator时需要，并且手动记录过了，所以在record里就不用再记录了，而且record也无法记录。

由于generator函数需要自己配置返回值、rsp等内容，还需要保证会执行f（arg），于是对\_\_ctx\_type也要做出修改。将原有的结构改为方便查看的3个void指针，分别保存返回地址，栈底，栈顶，增加一个4字节的形参。

在generator函数中，需要手动设置ctx的内容。将其rsp设置为开辟的栈空间的最大值-16（16字节对齐），将recover到时候返回的地址设置为函数f，再将参数置为arg，保证执行f（arg）。最后再把rsp+8中的返回地址置为throw函数，判断是否是generator正常结束调用就行了。如果是正常结束调用就需要throw配合抛出gen结束调用的信息。

最后在throw函数里增加一段判断是否是结束调用的情况。如果结束调用，此时异常处理栈就是空的，因为在所有的generator、yield、send中都不会对异常处理栈做操作。如果异常处理栈为空就说明调用结束，就该把当前栈的错误置为调用结束的标志，即宏定义中的ERR\_GENEND，然后恢复到generator到调用者的上下文即可。

