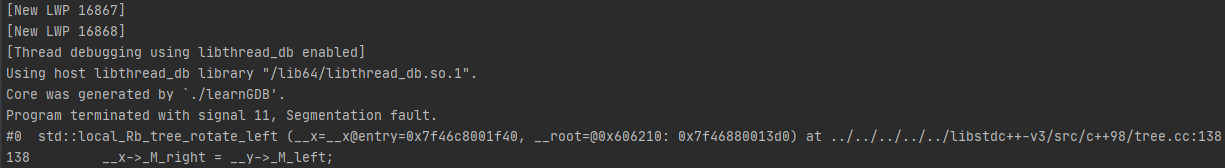
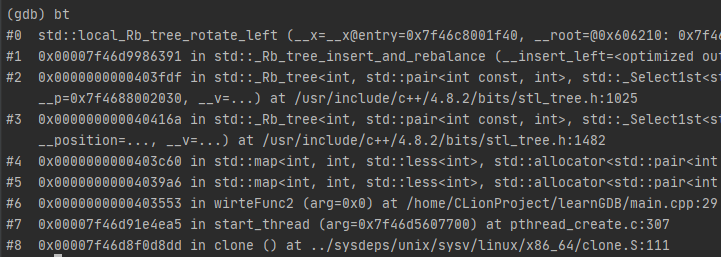
# 多线程读写map

## 错误及检查

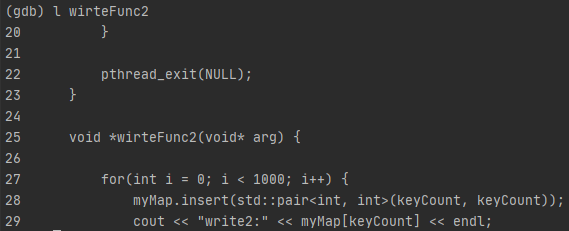
开启100个写线程，50个读线程，每个线程执行1000次写或读，终于出现了coredump，首先可以使用命令 gdb ./learnGDB core 进入gdb调试core文件，结果如下：



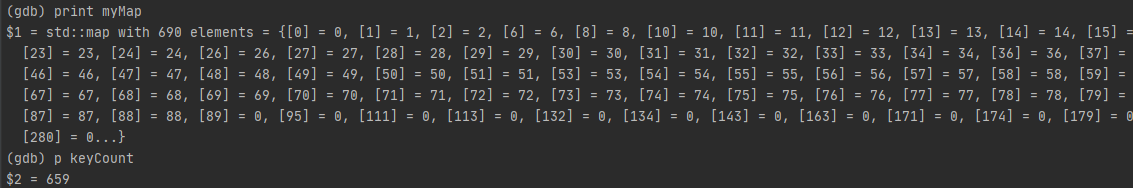
接着可以用bt查看堆栈，可以发现在main函数的第29行出现了错误



可以用 l wirteFunc2 查看具体出错函数



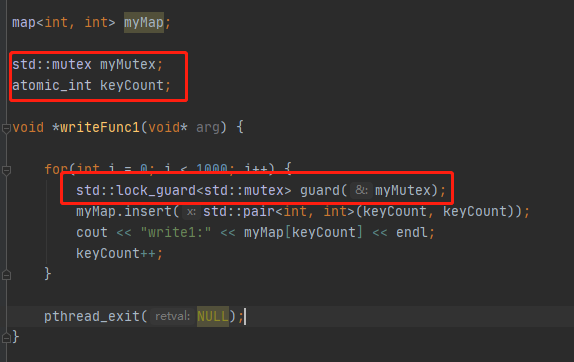
查看第29行的相关变量



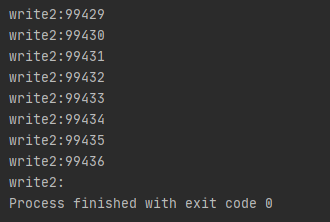
观察map的值和keyCount的值，发现插入的值从keyCount==89开始不对等，考虑是main函数第28行的map插入发生了错误，又因为wirteFunc2是线程函数，考虑是因为多线程写入map导致错误，可以在这个方向检查修改。

## 更改

可以在插入时加锁，并且将keyCount换成原子变量类型，如下：



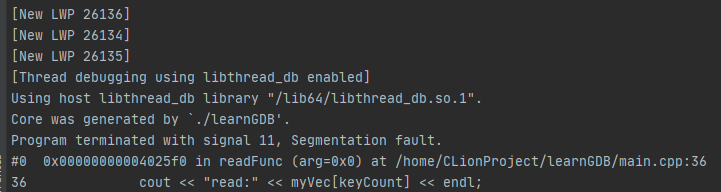
结果：上百个线程，每个线程成千次循环仍正常运行完毕



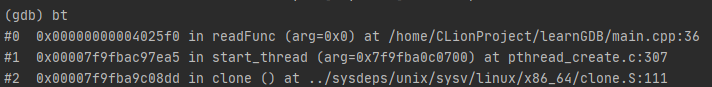
# 多线程读写vector

## 错误分析1

编写了一个不加锁保护的多线程读写vector（开启50个写线程，50个读线程，每个线程执行1000次写或读），出现coredump



bt 查看栈帧

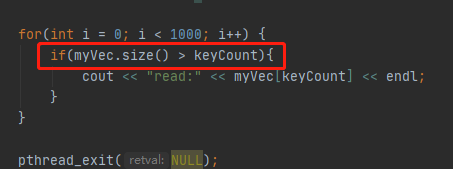


进入readFunc 查看相关变量



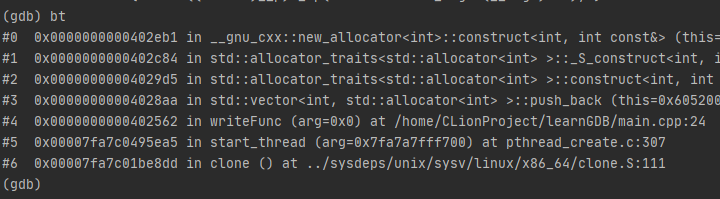


发现此时vector还没有容量，但是读函数已经尝试访问myVec[0]，即访问不存在的内存，所以发生了错误，因此要保证不能读未写的vector被读取。（写到这里发现前面的map读也要先确认值是否存在，否则段错误）

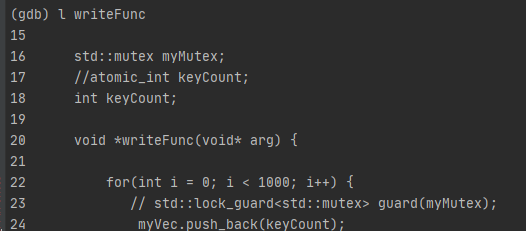


## 错误分析2

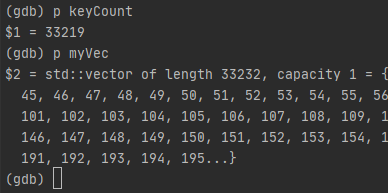
修改了前面的错误之后，发现还是出现错误，栈帧如下：



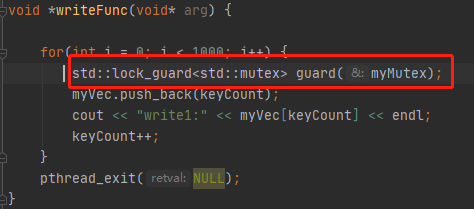
发现错误行



查看相关值：



与写map时类似，可以推测出是写错误，不同线程同时写同一个内存位置，导致coredump，应该在写时加锁，运行正常



# 其他

set的机制跟map基本一致，不重复分析了。

int的话，在上面的keyCount已经有体现，会出现的问题主要是多线程的自增冲突（两个线程同时获取这个地址，想要修改，发生错误）。如果需要保证线程安全，可使用原子操作atomic\_int，这个既可以保证修改时线程安全，又比加锁速度快，首选。

string的写vector这些类比较像，也是要在写的时候保证其他线程不占有该string实例，读的时候要保证该实例不是正在被修改并且该实例存在。