大家好，我是白天宇，今天由我来组织虚存的讨论。本该是这样，但是考虑到大家现在期末压力都很大，malloclab 又很难，所以我和助教商量后决定取消虚存的讨论，改为给大家稍微讲讲 malloclab 的注意事项。malloclab 还没有结束，我肯定不能给大家透露具体实现，所以我会给大家介绍介绍我 debug 的经验，一些我个人踩过的坑，以及可能的优化大方向。

malloclab 的 due 是下周一，deadline 是下周三，建议还没做的同学快点开始，不要觉得一个星期很长，这个 lab 是真的能让你盯着屏幕上的段错误看整整一天但一无所成的，这也就是为什么我要专门来讲 debug。当然，最好的情况是压根不要写出 bug，所以我先来说一个预防 bug 的建议：**不要好高骛远。**

我认识有一个大佬想一步到位，他直接写了个分离适配，然后去脚部、压指针、标记位等等所有技术全堆上去了，结构设计的很巧妙，写的时候很开心，一跑测试就傻了，bug de 都 de 不完，他改了整整三天，最后不得已，把整个代码废弃掉了。那个大佬是很强的，他都尚且如此，所以非常不推荐大家试图一步到位。

我的建议是一步一个脚印。比如你想写一个优化版的分离适配，先不要写分离适配，去写一个最基本的隐式空闲链表，什么优化也不要加。等到代码调通了，再改成显式空闲。再调通了之后，再改成普通的分离适配，再调通了之后，再逐个加你想加的优化。这样层层递进的好处是，每一次都只改动了部分代码，出错的概率更低，在 debug 时的怀疑对象也更少。相信我，慢就是快，多写个隐式空闲、显式空闲花不了多少时间，如果你尝试一步到位，在 debug 时就是草木皆兵，这是相当痛苦的。

当然，bug 是防不胜防的，所以 debug 的技巧也非常重要。malloclab 代码规模算是比较庞大了，肉眼 debug 是不现实的，这里我推荐一种 debug 方法：**魔兽式 debug。**

大家还记得程设的魔兽大作业吧，当时我们写了非常多的输出语句，通过与标准输出对比，就很容易锁定 bug 的发生地。我们现在没有标准输出，不过没关系，writeup 要求我们实现了一个 heap consistency checker，如果你实现了这个 checker，当堆的结构被破坏时，它就会打印错误并终止程序，这时候你再去找附近的哪句输出与预期行为不符，就能定位到 bug 的发生地了。

大家在 debug 之前一定要先实现一个详尽的 heap checker，写 checker 很麻烦也很无聊，但确实是有必要的。魔兽风格的输出语句也是一样，需要写很长时间，但还是那句话，慢就是快，绝对物超所值。

不过我们也确实可以再聪明一点，不用真的像魔兽一样写大量输出。我们都学过二分查找，找 bug 其实也是一种查找。比如我是这样做的，我会先在每个函数的开始输出一句“进入了某某函数”，这样当 checker 报告错误时，我就知道错误发生在哪个函数，并且知道整个调用链，然后我就在这个调用链里加更多输出，把 bug 锁定在某几句输出之间，不断缩小范围，直到定位到 bug 的精准位置。

写输出的时候还要注意一个非常关键的问题。虽然 `mm.c` 已经帮你定义好了宏 `dbg_printf`：

```
#define DEBUG
#ifdef DEBUG
#define dbg_printf(...) printf(__VA_ARGS__)
#else
#define dbg_printf(...)
#endif
```

但是我不建议这样使用。因为我们也学过，`printf` 是有缓冲区的，当段错误导致程序异常终止时，缓冲区并不会被清空，所看到的输出并不是所有输出，这会导致对错误发生地的严重误判，我就因此浪费了大量时间。虽然说是写入 `\n` 的时候会清空缓冲区，但我也确实遇到过多行未被输出的情况。原因不是重点，我们就不追究了，不过解决方法还是很简单的，比如可以将 `fflush` 作为 `dbg_printf` 宏的一部分，或者直接输出到 `stderr`，`stderr` 是无缓冲的。

另外，上下翻命令行还是很费劲的，大家可以直接把 debug 信息重定位到一个输出文件里，这都很简单，大家也学过 I/O，不用我教。

魔兽式 debug 适用于对小型 trace 调试，如果试图调试大型 trace，可能会导致输出文件过大。这里我也有一个技巧，就是先不输出调试信息地运行一次，并且让 heap checker 发现错误时输出错误块所在的内存地址。虽然 driver 是随机化的，但是地址后五位每次运行是固定的。得到这个特定内存地址后，我们就设置一个布尔变量，比如叫 `verbose`，初始设为 0，当检测到访问那个特定内存地址时置 1，并在代码中把所有调试信息改成当 `verbose` 为 1 时输出。这样我们就可以只关注错误发生地的上下文，而不会被前期冗余的输出干扰了。

当我们定位到错误的真正发生地之后，事情就比较简单了，大家也 de 过一年多 bug 了，单步调试、监控变量什么的应该是手到擒来了。我就提一点，如果你发现在一次 malloc 调用结束，另一个函数被调用之前，堆的结构遭到了破坏，那八成是 malloc 分配的内存过小了。driver 会在每次申请内存后写入随机位，如果实际分配的内存过小，就会破坏脚部乃至下一个块。

最后再简单说几句程序优化。隐式空闲和显式空闲的各种操作都是 $O(n)$ 的，而且几乎没有应对外部碎片的措施，显然不可取。建议大家以这两个为过渡，最后改造成分离适配。虽然也有一些人叫嚣最优解是 BST 或者红黑树之类的，不过那玩意常数巨大，目前来看最好的方式还是以分离适配为基础进行优化。

因为不能讲具体实现，我只能给大家一些方向性的提示：

- 去脚部：非常有用，但注意不要再访问已分配块的脚部，以及去脚部之前写的一些 `DSIZE` 要改成 `WSIZE`，我在这上面都吃过亏。
- 最小块大小：如果你一开始写出来是 24，可以尝试压到 16。16 已经足够好了。压到 8 很困难，在理论上是可行的，但没必要，几乎没有性能提升。
- 调参：调参是玄学，但也非常有用，尤其是当你会算卦的时候。我是说真的，我一开始是 97，我室友帮我算了一卦，改了个参数就 100 了。
- 面向 trace 编程：如果你发现某一个 trace 分数特别低，可以观察它的分配模式，从中获取灵感。但注意 writeup 说了，不能在程序中试图判断正在运行哪一组 trace，不然就过于不讲武德了。
- 时空平衡：malloclab 的评分并不是线性的，有时候差一点就会差好几分，我们可以尝试时间换空间或者空间换时间，以得到整体性能上的提高。以链表的结构为例：
    - LIFO 更快，有序链表在 first-fit 时空间利用率更高。
    - 单链表占用空间少，但删除结点是 $O(n)$ 的；双链表占用空间多，但删除结点是 $O(1)$ 的。
    - 指针一般用 8 字节存绝对地址；如果用 4 字节存偏移量，速度会显著下降，但节省一半空间。这里要特别注意，如果你用 4 字节存偏移量，一定要特判空指针。
- 常数优化：常数时间的优化肯定是效果非常不显著的，但如果你觉得算法层面实在无法提高了，也不妨试一试。Makefile 已经开了 O3，优化效果很好，但大家也不要迷信它，毕竟编译器是有局限性的。有些逻辑上的优化编译器做不了，例如把某个块插入链表，又立刻把它删除，只有人知道可以把两次操作合并。

我就讲这么多，可能讲得有点快，还没做 lab 的同学可能全程不知道我在说什么。没关系，我会把 PPT 和稿子都发在群里，等到大家段错误 de 到吐或者没有优化思路的时候可以看一看，希望能给大家带来一些帮助，谢谢大家。