Project 2 客观题杂谈

南京大学 孙博文
tilnel@smail.nju.edu.cn

谢谢选题的同学们，过程中出了一些波折，年前选做的同学被折腾得厉害，算是共同完成了客观题的出题完善工作。

分低不要担心，我会 curve 到和主观题相似的分布和平均分。分数过低的，我会看一眼完成度再给主观打分；ddl 前提交初始 repo 的同学，得准时与诚信分 20，看情况可能更高点。

认真独立做这两题的学习效果绝对不赖。

通过客观题的法宝就是充分测试。

Ramfs

ramfs 的测试点到 9 就算完结了。1 月 19 的晚上，sbw 躺在床上想破头也没想到如何构造与现实应用场景类似，又重 IO，又好用这几个捉襟见肘的 API 写出来的测试。于是干脆就直接取消了：反正 10 不管怎么写，估计都不会超过 1-9 的测试量了。

下面给出一些比较通用的建议。

输入合规检查

实现 API 的时候，要对输入数据进行 validate。test4 里的那个 rread(-100000000) 就已经足够惊悚，大家最后都改过来了。但如果在某一个数据点里偷偷再安插一个 rwrite(1000000)，你改正过一次的程序，也能应对这种摧残吗？...这种已经属于很初级的要求。但我们的数据点也并未再从边角料里挖两个坑给大家跳：手册里提过的那些边界情况，对了这次就对了。

管理内存

主张：分配的时候，就想好是谁最终释放。比如 ropen 中分配的空间，肯定是 rclose 释放；O_CREAT 分配的空间，最终肯定是 unlink 去释放。如果一段空间不需要跨函数使用，那么尽量做到谁分配，谁释放。

譬如，有这样的写法：

```c
char *generate_some_string(int arguments) {
    char *ret = malloc(1024);
    /* statements */
    return ret;
}

int ropen(char *pathname, int flags) {
    char *something = generate_some_string(args);
    /* statements */
    return fd;
}
```

malloc 的这段空间，由 generate 分配，主要由 ropen 实际使用，最终也只能在 ropen 中回收。但是孤立地看 ropen，并不一定意识得到它调用了一个会分配一段空间的函数而没有释放掉。所以，更好的做法是由 ropen 进行分配，将指针作为参数传给被调用者：

```c
void generate_some_string(char *destination, int arguments) {
    /* statements */
    return ret;
}

int ropen(char *pathname, int flags) {
    char *something = malloc(1024);
    generate_some_string(something, args);
    /* statements */
    free(something);
    return fd;
}
```

这样，free 和 malloc 就对应起来了。

管理内存的另外一点，就是注意函数的所有出口之前，释放掉用完的所有空间。有的时候函数中遇到某些异常状态，提前终止执行了，这个时候可能已经分配了一些内存。然而有的时候，只会想到在最终正常执行返回之前释放内存，疏忽了那些“分支”。

就算 coding 再怎么精打细算，也逃不掉释放不完的命。这种时候就要利用工具了：

address sanitizer。在 Makefile 的 gcc 后面加一个参数 `-fsanitize=address`，即可看到泄漏的内存总大小、泄漏位置。**是 Linux 专属武器**。

性能调优

实验评测的时间是充裕的，但仅限你没有写出非常消耗时间的垃圾设计。必要的循环遍历、链表遍历并不会让你得到时间超限。譬如，从 0 到 4095 循环来找到一个空闲的 fd；譬如，遍历子目录的 sibling 链表找到名称匹配的那一个。你不需要写节约时间（某种意义讲，也挺浪费时间的）高级数据结构，比如哈希表，etc。也不需要用倍增等方式去节约 realloc 的调用。

> Premature optimization is the root of all evil. --Donald Knuth

> Keep it simple, stupid.

讲一个我看到的案例。一位同学某测试点超时，我把测试点拆成两半，仅运行前半耗时 6s，仅运行后半耗时 5s。但如果放在一起，总耗时就会变成 90s：前半的时间不变，后半拖慢了很多。最终分析下来结果是这样的：前半执行时，他代码中的某个循环条件会增加。比如：

```c
for (int i = 0; i < cond; i++) {}
```

这个 cond 不断地增长。后半受到这个增长的影响，每次又都跑完了整个循环，时间就炸了。

但其实这是可以避免的！因为这个循环是"找到了就可以退出"的循环。他只是忘记写 break...当然他后来自己凭借肉眼找到了，我觉得很牛逼。

除了肉眼，我们还有什么办法做性能测试呢？当然一切的前提是：自己写了测试点。

以 789 三个点为例，7 是大文件测试，那少说也得：

```
1   rwrite(fd, buf, 400 MB);
```

像这样顶着上限去测试。

8 是文件树测试，这就更容易了，无非就是宽度和深度。

9 是 fd 管理。那就是频繁 ropen, rclose，以及多个 fd 一起读写。OJ 测试的上限是 4096，自己测就往极限去靠就 OK 了。

有了测试点以后，我们就可以运行了。测试点在设计上，要能够灵活地修改规模。在不同规模下，对同一性质的测试点做多次测试，可以总结一些规律。也可以对测试的运算量的数量级以及运行时间做预测。如果结果比预测慢很多。就可以去被测函数的代码里寻找性能下降的原因，比如 fd 资源紧张因此分配变慢，这是合理的；但 fd 资源空闲时依旧分配较慢，这就不太对劲。

> 目前大多数个人计算机的效率在每秒 1e8 次循环的数量级。

有什么工具能定位性能问题？

Linux Perf.

运行错误

老规矩，先用测试把问题暴露出来。在此基础上，gdb 也好，IDE 的断点调试、内存监视也好。

Gitm

gitm 目前有一位同学选做，做得还是很用心的。估计后续没人想选做，就摸了。这题比起 ramfs，系统设计的味儿更浓一点。

这题的原型是去年 Project-2 的 git--。我去年是期待出成今年这个形式的，最后原操刀人出成了标准输入交互程序。大家都喜欢写熟悉的 OJ，看到是客观题就被骗进来，但是因为难度不低，最后分数惨淡，不得不 curve。Ramfs 还算离传统 OJ 的形式近一点；再近一点，无非是我把函数调用改成格式化的输入，用 scanf 代替编译链接 main.c。设计成这样就是为了让大家不得不学学 Makefile，命令行这些东西。gitm，就走得更远了，它要求你的程序和自身以外的更多东西打交道了：参数解析，数据持久化，外部命令...整个程序的逻辑非常好想，但是搞明白每一样东西怎么用，怎么将它们配合起来，就比较困难啦。

这题其实是有捷径可以走的。只要会组合一点命令行，配合 diff 和 patch，就能轻松写出一个 200 行以内的形 C 实 shell 的 gitm，能得到 80 以上的分数。

测试思路

difftest：用一个正确的实现和你的实现做对比。

ramfs 的正确实现这不是就很好找吗？把 ramfs 的 API 用真实文件系统 API 去实现，这就是标准实现啦，至多需要把两者的差异做一个适配，写下来也就没几行代码。这样的标准实现，首先它可以用来证明你的测试写得没问题。

注意 Linux 默认进程打开的文件描述符上限是 1024，并且开局已经占去了 3 个。但是这也够你自己的实现喝一壶的了。

其次就是用来做 difftest：你做一步，我做一步，然后我们看看结果是不是一样呗？

gitm 也是类似的思路。这不是有 git 可以参考嘛，我们只需要两边各做一步，然后对比结果，就好了。。。记得用脚本而不是傻乎乎地每次都手敲。

扩充知识

全是关于 ramfs 的。但不能帮助你写 ramfs。

Linux 文件系统 API 的实际情况和 ramfs 有差异：

open 其实不支持 O_WRONLY | O_RDWR，会打开失败。

使用 O_APPEND，不管 rseek 到哪，永远都是在结尾 write。此时 offset 只能对读操作生效。

打开已有目录时，O_CREAT 会导致失败。

unlink 时如果文件仍在打开中，会把 unlink 交由 fd 来做。即最后一个引用文件的 fd 被关闭时，文件（的这一链接）被删除。

> 什么是链接？为什么叫 unlink 而不叫 remove、rmfile？

文件只是节点。这一节点可以在多处被多次引用，这就是链接的实质。对同一文件的多个硬链接只占1份空间。文件系统管理时使用“引用计数”，只有在引用计数归零的那一次 unlink 时，文件内容真正从文件系统中消失。

目录是可以 open 的。我们不能 read 或 write，但是可以 openat（在 fd 指向的目录中打开文件），getdents（获得 fd 指向目录中的子目录项）。

总结

人生苦短，多用工具。

有 bug 不要盯着看。如果本地跑过了，就想办法写点测试让它跑不过；如果本地跑不过，但是看不出问题，就跑起来看（调试器）。

内存泄漏和段错误有 sanitizer；答案错误可以打 Log 看看是哪一步出问题；时间超限有 perf。

学习新的工具，每样学 10 分钟，能节省 90% 的瞪眼时间。浪费的时间都够读个 PhD 毕业了（别骂了）。