实验报告-PA3

甘晨 181240014

2020 年 1 月 2 日

此次实验未能按时完成，迟于 DDL 提交。

1 实验进程

1.1 PA3.1

实现了异常响应机制，事件分发和上下文抽象。

1.2 PA3.2

实现了加载器，相应的系统调用和堆区管理，并成功运行了 Hello World。

1.3 PA3.3

实现了文件系统，以及将设备、串口和 VGA 抽象成文件，成功运行了《仙剑奇侠传》，最后实现了开机菜单程序。

2 必答题

2.1 文件读写的具体过程

首先回顾一下各个部分的功能：

NEMU: 模拟出一套计算机的硬件，尤其是 CPU 的功能

AM: 为程序的运行提供运行时环境，提供软件功能，以便执行更加复杂的程序

库函数: 程序对运行时环境的需求的集合，库函数可以通过 API 被程序使用，来达成程序的需求

Nanos-lite: 实现操作系统的功能，例如程序加载，异常响应等

libos: 提供各个系统调用的功能

读取存档

首先 PAL_LoadGame() 函数调用了 fopen, fread 和 fclose 来打开读取和关闭文件，这写函数定义在 Nanos-lite 中，通过 libos 封装并提供接口，调用 libos 中相应的函数，实际上最终时通过系统调用调用了在 Nanos-lite 中定义的函数来实现文件操作。当然，对于文件的操作需由操作系统进行，属于操作系统内核区的运行时环境，需要陷入内核进行。在这一过程中 AM 和库函数提供了程序需要的功能，例如输入输出，虚存管理和在陷入内核时的上下文管理，而 NEMU 作为硬件支撑，提供了 CPU 的指令支持以实现上述功能。还可以发现，在 PAL_LoadGame() 函数中调用了 memcpy 和 memset 库函数，这也是由 AM 所提供的运行时环境。PAL_LoadGame() 将存档读入 PAL_LARGE SAVEDGAME 类型的文件 s，然后将其赋给 LPGLOBALVARS 类型的 gpGlobals，这实际上就实现了读取游戏存档。关于 fread(), 正如前面所说，这需要陷入内核执行，其中触发异常响应后，Nanos-lite 中的 do_event() 函数会识别出系统调用事件，然后根据系统调用号调用相应的函数，这一步在 libos 中会将相应的参数存入寄存器，供 syscall.c 中相应的函数使用，在 do_syscall 中最终会调用文件系统的 APIfs_read()，最终完成读取存档文件的操作。

更新屏幕

更新屏幕的操作与 VGA 的操作本质上是一样的，基于已经实现的虚拟文件系统，显存已经被抽像成了文件，因此写显存的操作与写文件的操作几乎没有差别，具体的各个部分的协同运作关系也与上面的读取存档类似。NDL_DrawRect() 函数是由 Navy-Apps 封装的多媒体库，实际上也可以归结为 AM 提供的 API，正如 PA 讲义中所指出，基于这一 NDL 库函数，用户程序可以实现 I/O 操作。我们在实现把 VGA 显存抽象成文件时，根据 Nanos-lite 和 Navy-apps 的约定，支持了对显存文件/dev/fb 和/dev/fbsync 的写操作，从而实现了刷新屏幕，而阅读 NDL_DrawRect() 函数可以发现，这一函数最终调用了 fwrite() 来写屏幕，最终达到了更新屏幕的效果。所以说，通过虚拟文件系统的抽象，把设备也抽象成了文件，伴随这对文件操作 API 的拓展，也就是 AM 的拓展，就可以运行更复杂的用户程序。

3 实验心得

3.1 PA3.2

PA3.2 的实验过程中踩了几个巨坑，有一个 bug 甚至干掉了一周的时间。第一个 bug 是在实现加载用户程序时，在终端 make run 之后，会输出许多 Log，最后一条会输出 “Jump to...” 表示跳到了用户程序中执行了，但我遇到的问题确是 nemu 会将刚刚输出的 Log 从头再输出一遍，这样不断的循环最终造成死循环。我最初觉得可能时 loader 不对，但 Log 中输出的跳转地址确实是 elf 头里的 e_entry，但有点不放心，借用了大佬代码替换调试了下，发现还是错的，这一问题排除。摸不着头脑的我开始回去重做 3.1 的任务，发现都没有问题，应该不是 3.1 里的坑。这时候心态有些爆炸，利用 diff-test 测了下，发现 pc 跳转到了一个 qemu 无法追踪的地址。于是开始利用基础设施，追踪

到所谓的跳转到 e_entry 之后的状态。令人震惊的发现，pc 跳转到了地址为 0 处，然后 pc 不断增加，一直增加到 0x100000 的 start 处，重新开始执行一遍，这才造成了我所遇到的死循环。后面经过定位，发现始 int 80 的问题，这导致了 pc 变成 0。这时候回去检查了 raise_intr 和 lidt 指令，通过输出调试法，发现根据 NO 索引得到的中断门描述符取到的地址为 0。那么为什么地址为 0 呢？面对这一问题，一时感到有些无解，但隐隐的感觉到可能时什么东西把中断描述符表给冲刷掉了。后来，在大佬的点拨下，我尝试在利用之前的 _yield 操作，来定位冲掉中断描述符表的操作，反复实验后发现，实在 proc.c 文件中的 init_proc() 函数中的一条 Log（“Initializing process... “）指令。因为在这条指令之前调用执行 _yield，在 raise_intr() 里通过输出调试法输出的中断服务程序地址不为 0，而在这条指令之后调用 _yield，就发现中断服务程序地址变为 0 了。再接着定位发现 Log 调用的是我自己的 print 库函数，检查发现我在 vsprintf 中定义了一个大小为 65535 的字符数组，而一个页的大小是 4KB，比页还大。将数组大小改小之后，问题就解决了。由于在 PA2 中，我已经被 printf 坑了一次，并做了修改，没想到还有坑，但是上还没有结束，后面我还被 printf 坑了一次。第二个不能成为 bug，主要源自我未理解框架代码。在实现识别 _EVENT_SYSCALL 和 SYS_exit 时，为了验证识别成功，我模仿了 _EVENT_YIELD 中输出一句话，执行程序却发现没办法结束程序，好像在哪里卡住了。通过定位发现，程序最后执行了一条跳转到该指令所在地址的指令，一直在哪里循环。我以为是我的实现上出了 bug，开始摸不着头脑的调试，发现好像没什么办法，于是去查看了 nanos.c 的代码，结果发现 _exit() 确实最后就是一条 while(1) 指令。恰好当天下午蒋炎岩老师在习题课讲到了有关进程的概念，原来进程不可随意返回，往往以死循环结束，恰好映证了我所遇到的问题。

3.2 PA3.3

PA3.3 的 events_read() 的实现过程中遇到了一个难以理解的 bug，就是加载运行/bin/events 后会输出各种奇怪的随机符号，而不是事件信息，有了 PA3.2 中的经验，我猜测可能是 printf 的问题，于是再去调整数组大小，我试着修改了最大的那一个数组的大小，结果随着大小的改变，输出也有所改变，在某一个大小范围内，能实现分行输出，事件信息为随机符号；在令一范围内，也能分行输出，但事件信息为空；有时不能分行输出。在做了相当的修改尝试未果的之后，借鉴了大佬的代码，结果发现 events_read() 成功了。于是，我这里暂时利用了大佬的 printf，在这一基础上继续了后面的实验，准备后面再对自己的 printf 做修改。不过还是想说，输出调试法还是厉害！在 PA3 中，通过替换调试法最终定位到了 PA2 没法跑 mario 的原因，是在于减法指令实现错误，应当是相减的结果直接送入 dest，我多了一步符号拓展，事实上符号拓展实在生成标志信息时才需要，只要先把结果送 dest 就好了。

4 鸣谢

由衷感谢给予我帮助和指点的几位大佬，在此表示深深感谢。