# ics-PA3实验报告

高辰潇
人工智能学院
SID：181220014

## 一、实验进度

- 完成了PA3中所有任务，并实现了difftest开关、快照等部分选做题

## 二、必答题

1. **解释行为：在navy-apps/apps/pal/src/global/global.c的PAL_LoadGame()中通过fread()读取游戏存档**
   - **app**：PAL_LoadGame向fread传入四个参数，&s，sizeof(SAVEDGAME), 1和fp, 表明从文件指针fp中读出sizeof(SAVEDGAME)到结构体s中。fread最终会调用navy-apps/libs/libos/src/nano.c文件中的_read， 借助_syscall_函数生成系统调用的汇编代码并将系统调用参数保存在寄存器GPR2、GPR3和GPR4中，将返回值保存在GPRx中。
   - **am**：随后运行程序，在程序运行到该汇编指令时，会从opcode识别出这是一条int指令，然后依次保存上下文，并根据索引0x80找到自陷的目标地址。跳转到该地址并执行一些相关指令后，会继续调用__am_irq_handle函数。__am_irq_handle函数将本次调用识别为_EVENT_SYSCALL，然后调用user_handler函数对该事件进行处理，此时系统调用进入操作系统层面。
   - **nanos-lite**：由于初始化时user_handler被初始化为do_event函数，因此最终nanos-lite会使用nanos-lite/src/irq.c中的do-event来对该事件进行处理。并且由于事件类型属于_EVENT_SYSCALL, 因此会进一步调用nanos-lite/src/syscall.c中的do-syscall函数。do-syscall函数对不同的系统调用进行分发，从上下文中读出参数，调用参数并设置返回值。我在实现该部分功能时将每种调用需要进行的操作进一步封装成了函数sys_*，例如SYS_read对应的操作为

```
size_t sys_read(_Context* c) {
int fd = c->GPR2;
void* buf = (void*) c->GPR3;
size_t len = (size_t) c->GPR4;
return fs_read(fd, buf, len);
}
```

   即使用fs_read读取存档内容到目标位置，并将它的返回值设置为系统调用的返回值，该返回值将一路返回到应用程序中。至此，完整的系统调用便已完成。
2. **解释行为：在navy-apps/apps/pal/src/hal/hal.c的redraw()中通过NDL_DrawRect()更新屏幕**
   - **app**： redraw首先生成图像信息fb，然后调用NDL_DrawRect函数。该函数最终会调用navy-apps/libs/libos/src/nano.c文件中的_write， 借助_syscall_函数生成系统调用的汇编代码并将系统调用参数保存在寄存器GPR2、GPR3和GPR4中，将返回值保存在GPRx中。
   - **am**：随后运行程序，在程序运行到该汇编指令时，会从opcode识别出这是一条int指令，然后依次保存上下文，并根据索引0x80找到自陷的目标地址。跳转到该地址并执行一些相关指令后，会继续调用__am_irq_handle函数。__am_irq_handle函数将本次调用识别为_EVENT_SYSCALL，然后调用user_handler函数对该事件进行处理，此时系统调用进入操作系统层面。

- **nanos-lite**：接下来的过程和问题一中的过程完全相同，区别在于，由于系统调用类型为SYS_read，因此会调用我所封装的sys_write进行处理

```
size_t sys_write(_Context* c) {
int fd = c->GPR2;
void* buf = (void*) c->GPR3;
size_t len = (size_t) c->GPR4;
return fs_write(fd, buf, len);
}
```

sys_write会从上下文中读取出文件标识符等参数，并使用fs_write向目标文件写入。由于目标文件/dev/fd并不是“文本文件”，因此fs_write实际上会调用VFS中的fbsync_write函数，使用am中的IOE硬件接口写入vga信息。至此，完整的系统调用便已完成。

## 实验中遇到的bug

- 实现loader函数部分，我在加载段内容到内存中这一步骤上遇到了溢出的问题。在读取每个需要加载的段的内容时，我一开始使用了一个字符数组buf来存放从elf文件中读出的内容，然后将buf的内容写入到内存中。由于需加载的内容较大，我将buf的大小定义为50000。然后，在运行hello.c时我发现int自陷后跳转的目标地址不太对劲，也就是IDT的内容被篡改了。找了很长时间之后...最终才发现是loader中这个大小定义过大的数组的锅。在向这个数组写入内容时，由于数组的大小过大，写入的内容把IDT的内容给覆盖了，导致了后续过程中的一系列bug。最终在确保不会出现安全事故的情况下，我将buf的大小重新设置为10000，循环读出elf文件中的内容并写入内存。
- 尚未解决的bug：在运行text和pal时，尽管程序都能正常运行并且都能得到正确结果、画面正常显示，但在difftest时，在一条movl 0xc(%esp), eax指令处DUT和REF的eax寄存器的值却不相同。很明显这个问题是因为内存中存放的值不同而引起的。不过，在与一位同样使用ubuntu的同学交流后我发现，他在进行difftest时也遇到了这个问题——甚至出现问题的指令、eax寄存器的错值都一模一样，我怀疑这个bug可能和使用的linux发行版、gcc的不同版本有关。目前我还在寻找这个bug出现的原因。