Programming Assignment 4 实验报告

黄奕诚　161220049

2017-12-20

一. 实验进度

完成各阶段所有任务，实现了分页机制、分时运行及时钟中断。

二. 必答题

不妨从入口函数_start 开始，逐步分析 pal 与 hello 程序的运行情况：

```
__attribute__((section(".text.unlikely"))) void _start(int argc, char *argv[
], char *envp[]) {
  int ret = main(argc, argv, envp);
  exit(ret);
  assert(0);
}
```

直接执行 main 函数，并且不会返回。在 main 函数中，

```
load_prog("/bin/pal");
load_prog("/bin/hello");

_trap();
```

分别对 pal 和 hello 执行了 load_prog，然后触发_trap。首先从 load_prog 谈起：

```
void load_prog(const char *filename) {
  int i = nr_proc ++;
  _protect(&pcb[i].as);
  uintptr_t entry = loader(&pcb[i].as, filename);

  _Area stack;
  stack.start = pcb[i].stack;
  stack.end = stack.start + sizeof(pcb[i].stack);

  pcb[i].tf = _umake(&pcb[i].as, stack, stack, (void *)entry, NULL, NULL);
}
```

nr_proc 统计了进程的个数，pcb[]数组记录了每一个需要运行的进程，通过对 pal 和 hello 执行 load_prog，将它们都置入 loader 函数中，分别加载，而 loader 函数在实现完分页机制后已经可以支持两个程序映射到不同的地址且同时存在。此时分页机制便承担了它的使命（分页的具体的执行过程在讲义中已经阐述得比较详细）。

再谈 load_prog 第二部分：

```
  _Area stack;
  stack.start = pcb[i].stack;
  stack.end = stack.start + sizeof(pcb[i].stack);

  pcb[i].tf = _umake(&pcb[i].as, stack, stack, (void *)entry, NULL, NULL);
}
```

前三行代码是为_umake 函数提供参数的，_umake 在 AM 中，用来创建用户进程的现场，初始化陷阱帧中的每一个域，用来实现不同进程之间的切换，现场的保存与恢复。而 main 函数中的_trap（在 AM 中）只是用来手动触发一个 0x81 的事件，来验证_umake 实现的正确性。

触发事件需要 int 指令，而 int 指令执行之前需要 lidt 指令，这都是在 nemu 中落实的。

再谈硬件中断，它的实现与 0x81 如出一辙，因为它与硬件相关，所以主体代码体现在 nemu 而不是 nanos-lite 中。但很重要的区别体现在了 exec_wrapper 函数中：

```
  if (cpu.INTR & cpu.IF) {
    cpu.INTR = false;
    raise_intr(TIMER_IRQ, cpu.eip);
    update_eip();
  }
}
```

而 cpu.INTR、cpu.IF 的设置已在其他地方有序地设定：

```
void dev_raise_intr() {
  cpu.INTR = true;
}
```

在 raise_intr 中，有`cpu.IF = 0;`

由此可以在每次执行完一条指令后对中断信号进行检查，使计算机的运行更加具有灵活性。

三. 实验过程中遇到的问题

a) 内核映射怎样实现？需要调用什么 APT？

当时尝试_map 函数的实现花了不少时间，出现各种编译错误，而且也忽略了 present 位为 0 的情况（这是助教的 slides 上指出的……）后来仔细翻了下 ICS 的书以及阅读了 x86.h 等源代码，理清思路后实现了出来。实验讲义上描述如此简略的原因大概是鼓励我们 RTFSC 吧……

b) 为什么明明成功实现了_map 以及 mm_brk，却仍然会跑出地址越界？

这是个被同学提醒之后才发现的问题，原因是 syscall.c 中，SYS_brk 中没有调用 mm_brk 函数……当时 debug 比较郁闷，后来突然一想：谁去调用 mm_brk 函数呢？然后发现了端倪。

c) _make 函数要初始化的陷阱帧具体是怎样的？

讲义上给了这张图：

```
|               |
+---------------+ <---- ustack.end
|  stack frame  |
|   of _start() |
+---------------+
|               |
|   trap frame  |
|               |
+---------------+ <--+
|               |    |
|               |    |
|               |    |
|               |    |
+---------------+    |
|       tf      | ---+
+---------------+ <---- ustack.start
|               |
```

stack_frame 应该指的是_start 的参数以及它的返回地址，而 trap_frame 感觉有点迷，后来想到之前 PA3 实现异常处理机制时定义的一个 trap_frame，发现果然就是这个。

d) **实现上下文切换的时候感觉很多东西讲义上没说？**

刚开始写 PA4-2 的时候，有个大佬提醒我这儿有坑，后来我发现其实不是坑，如果没有发现 trap.S 要实现 vertrap 函数，在 irq_handle 要添上_EVENT_TRAP 的情况，在_asye_init 函数中也要相应添加，那只能说明 RTFSC 还不够……因为看到 0x81 这个事件的时候应当自然而然想到之前实现的 0x80 系统调用事件，从而考察每个与 0x80 有关的代码位置。讲义上虽然没明说这些地方要改，但都暗指了：

能与ISA相关，是由ASYE的 `_trap()` 函数提供的. 在 `x86-nemu` 的AM中，我们约定内核自陷通过指令 `int $0x81` 触发. ASYE的 `irq_handle()` 函数发现触发了内核自陷之后，会包装成一个 `_EVENT_TRAP` 事件. Nanos-lite收到这个事件之后，就可以返回第一个用户程序的现场了.

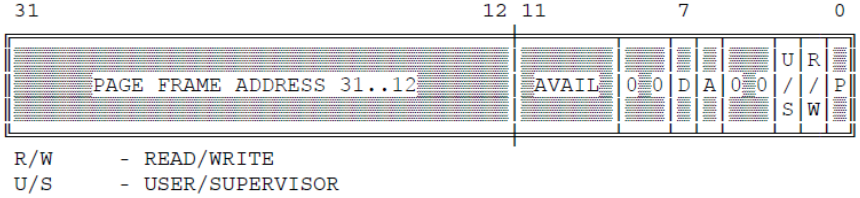
这句话因为不在红框中，相对来说容易被忽视，但它其实是浓缩的精华……

四， 选做题

a) **为什么表项中的基地址信息只有 20 位，而不是 32 位？**

以下是分页机制的表项：

Figure 6-10. Protection Fields of Page Table Entries



b) **为什么不采用一级页表？或者说采用一级页表会有什么缺点？**

一级分页一定会在占用 4M 的内存，因为目录项一定会全部加入到内存中；而二级分页占用内存少，因为 4GB 内存没有被全部分配数据时，这是页表就不会被调度到主存中；

c) **关于优先级调度**

如下修改代码（对 pal 和 hello 区别处理）就可以实现 1000:1 的调度

```
_RegSet* schedule(_RegSet *prev) {
//  printf("schedule!\n");
  current->tf = prev;
  if (first_process) {
    current = &pcb[0];
    first_process = false;
  }
  else if (current == &pcb[1])
    current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
  else if (process_cnt>=1000 && current == &pcb[0]){
    current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
    process_cnt = 0;
  }
  process_cnt++;
  _switch(&current->as);
  return current->tf;
}
```