OS第 3 次实验

进程管理

151220030 → 高子腾 : GZT@outlook.com

2017 年 5 月 5 日

谨以此实验报告纪念... 算了不想纪念了

https://github.com/sebgao/sebopesyslab

不经意间已经被 clone 十三次了，哭笑不得

第一部分 实验环境

编译虚拟机环境 Debian 32bit，注意只能在 Debian 32bit 下编译的镜像文件才能正确运行（不能在 Ubuntu 32bit 下编译）. 运行 QEMU 的平台则没有限制.

GCC 编译版本 gcc version 4.9.2 (Debian 4.9.2-10)

第二部分 实验结果

完成了所有要求，

1. 能编译运行游戏
2. 实现 fork()
3. 实现进程调度
4. getpid(), sleep(), exit()
5. 有 git 纪录

第三部分 详细实验结果

1 分页技术

分页的 pmap.c 和 entrypgdir.c 具体细节我是参照 JOS 的实现方法的. 由于 JOS 采用固定分页制，所以一进入 kernel 的时候就需要汇编打开分页机制，这部分代码可以在 kernel/src/entry.S 里找到，由于编译的时候认为是已经分页机制开启了，所以在 entry.S 的分页之前的符号引用时都需要 RELOC 处理，将高位的虚拟地址转换为低位的物理地址. 将写死的页目录加载到 cr3，然后就可以开启分页愉快的玩耍了.

2 PCB 组织结构

我的 PCB 组织结构挺怪的，主要是因为我有两种类型的进程，一种是在 kernel.c 写好的函数产生的进程（没错，内核代码段是可以产生进程，而且可以用所有系统调用），我称之为内核级进程，一种就是普通的用户级进程.

```
typedef struct PCB {
  uint8_t kstackbottom[0x10];
  uint8_t kstack0[NPKSTACKSIZE];
  uint8_t kstack0top[0x10];
  uint8_t kstack[NPKSTACKSIZE];
  uint8_t kstacktop[0x10];
  uint8_t kstackprotect[0x10];
  struct{
    uint32_t used;
    uint32_t pid;
    uint32_t ppid;
    TASK_STATE ts;
    TASK_TYPE tt;
    uint32_t timeslice;
    struct TrapFrame *tf;
    pde_t *pgdir;
  };
  struct PCB *next;
  struct PCB *tail;
} PCB;
```

其中关于栈的大多数部分是用来定位和保护的，最主要的还是 kstacktop，kstacktop 是用来

做用户进程的内核栈栈顶的，也是用来做内核进程的普通栈和内核栈栈顶的（虽然有点绕口，但就是这么用的），这里需要注意的是，内核进程由于 cs 一直是特权级（不然就没法执行 kernel.c 里的代码），所以栈不会切换，因此内核进程的内核栈和普通栈是公用的，所以有这么拗口之前的一段话.

这里还有一段血泪史，我一开始没注意到栈是往下长的!! 所以怎么调怎么不对. 还有就是不能直接把 kstacktop 的指针作为用户栈顶，这样做会栈溢出，这个坑更加隐秘，所以 kstackprotect 出现了!

used 指示这个 PCB 有没有被使用掉，这个很好理解.pid，ppid 不赘述了. ts 指示任务状态，但我的系统并没有用太多 ts 相关的内容，原因是进程调度都是用链表的.tt 指示进程类型，是内核级进程还是用户级进程.timeslice 很好理解了，是时间片.tf 是陷阱帧,pgdir 是页目录. next 和 tail 是有关调度链表的指针.

3 实现 fork()

实现了页目录深拷贝后实现 fork (kernel/src/process.c) 就简单多了，不过还是有几个坑的:

1. 正如张枣在《镜中》一诗中所言，

   “一面镜子永远等候她，让她坐到镜中常坐的地方”

   新 PCB 的内核栈永远等候着新的 tf，新的 tf 必须指向新的内核栈相同偏移量的位置上，不然后果很惨烈!

2. 同上的原因，内核级进程由于 cs 是特权级，没有 esp 的改变，内核栈与用户栈共用，新的 tf 中的 ebp 必然指向旧的进程的栈中，所以对于新的进程，必须要改所有栈帧的 ebp 值，详见 kernel/src/process.c.

4 实现其他调用

1. 对于 getpid()，getppid()，实现起来很简单，也没有什么坑.

2. 对于 sleep()，这里复用了 timeslice,sleep 时设成将 timeslice 设成频率乘以参数，切换到 sleep_list 链表中去，每次时钟中断时将 timeslice 减一，当 timeslice 减到零时，将它移到 ready_list 就绪队列中去.

3. 对于 exit()，就是物理页的释放,PCB 的释放，没有什么大的障碍.

5 关于进程调度的一切

关于进程调度的一切都可以在 kernel/src/scheduler.c 中找到，思路非常清晰，只是实现链表函数非常麻烦，花了我一晚上.

第四部分 综合

本着有什么就展现什么的原则，我在代码里加了很多功能测试的模块，听我娓娓道来:

进入 kernel 的 main 之后，分别会初始化三个进程,pidle，pcb，pcc. pidle 是空闲内核级进程（以 kernel.c 中的 idle 函数为入口，死循环里不断出让 CPU），他的存在纯粹是测试调度算法. 而 pcc 是测试一系列功能的进程（以 kernel.c 中的 busy 函数为入口），他会先连续 fork 三次，产生 8 个子进程. 随后，进入循环，通过 yield 出让 CPU 资源,sleep 相应 pid 的值，在三次 sleep 唤醒之后，通过 exit 结束进程.

pcb 就是游戏所在的进程，当然按照要求他先被当作 init,fork 出游戏之后 exit 结束. 而在游戏开始之后，可以看到右上角是 pid 的值，如果看不清楚，可以按 P 键放大观察. 而按 L 键可以使游戏进程 sleep 五秒，这五秒内游戏不会有任何响应，但唤醒时，游戏逻辑会追赶时钟，会让你体验到前所未有的快感（误）(画面不会被清空，因为其他进程没有用到显存）. 你也可以按下 M 键调用 exit 来结束这无聊的游戏，当然如果你是在其他进程结束之前按 M 的话，可以看到游戏结束之后其他进程一样输出.

附游戏操作指南备忘录：欢迎画面按 Q 进入游戏，按 WASD 控制蓝球，按 LMP 执行相对应的特殊功能

第五部分　必答问题

上次实验的分页机制问题：

1. 如果你参考 jos 的代码，在 entry.S 中，有这样 2 行代码：

```
_start = RELOC(entry)
...
mov  $relocated, %eax
jmp *%eax
```

答：　由于一开始没开启分页机制，但编译器已经把程序相关的符号重定位到 0xc0100000 高地址上，而没开启分页的程序需要的是真正的物理地址，所以进行 RELOC 处理，减去 0xc0000000 得到 kernel 符号的物理地址 (boot 加载 kernel 时加载到物理地址 0x0100000 上). 而 jmp 时 eip 还处在低位，为了在进入 main 函数之前进入高位就 jmp 到 relocated 符号的值上了.

2. 你应该注意到了，我们的数据结构中没有存放页框的物理地址，那如何根据一个 page_info 元素找到它对应的页框呢?

答：　直接计算 (current_page_info-&(arr_page_info[0]))«12，current_page_info 指针指向当前 page_info 元素，arr_page_info 是 page_info 数组.

3. 能不能根据物理页的起始地址找到它对应的数据结构呢?

答：可以，arr_page_info[pa»12]

4. 你需要将 kernel 的页目录，拷贝一份作为进程的页目录的模板，这样做为什么可以，会不会带来什么问题?

答：复制 kernel 的页目录项，其实就是为子进程提前做好内核区的内存分配问题，因为内核区的内存数据是共享的，所以这样做并不会带来什么问题．相当于每个子进程都有共享的内核代码段内核数据段.

第六部分　实验心得

这次实验算是这个学期遇到 bug 最多的项目，在一布置实验后我就着手工作，最后还是整整花了两个周末才弄好基础性的架构．因为上次实验的分页偷懒没有写，相当于给这次实验挖了一个很大的坑，而且分页机制又缺少指导，研究别人写好的 JOS 代码才有点点收获．虽然过程还是蛮艰辛的，但是收获还是很大的！一些 trivial 的细节我在写 lab 时竟然会忘记，比如栈是往下长的这样一个非常小的细节，为此也耗费了许久精力．因为我在两周内基本写好了，所以很多人向我请教问题，还是蛮骄傲的

　　ヽ（•ω•）ヽ