Lab5 实验报告

- 于凡奇 18307130182

问题一

trapframe 和 context 的实例都存在栈上，它们的具体地址在进程初始化之前无法确定，只能通过指针引用。这样设计也可以节约内存空间，process 列表中未使用的 proc 不需要无谓地占用相应的空间。

问题二

由于我们的 `context` 直接存到栈上，其起始地址是不能指定的，而是由上下文切换前栈的状态（栈指针位置）决定。`swtch` 的调用者想要得到旧 `context` 的位置，只能给 `swtch` 传递一个二级指针，让 `swtch` 将旧 `context` 的地址存放于二级指针指向的位置。

`swtch` 作为一个（相对）正常的函数，其调用是由正常的程序控制流产生的，其调用者有义务保存自己需要的 caller-saved registers，而被调用者只需要保存恢复 callee-saved registers。这是 ARM 的编程规范，能够减少不必要的额外的用于保存状态的开销。

陷入有可能是异步事件，此时原本的程序控制流被强行打断，还未来得及保存自己需要的 caller-saved registers 就已经失去了 CPU 的控制权。在这种情况下，操作系统就需要替被打断的程序保存这些额外的状态于 trapframe 中。

（未完待续）

流程介绍

下面按照第一个用户进程的初始化顺序介绍使用到的主要函数。

proc_init

初始化进程表相关，目前只有 `ptable.lock` 。

proc_alloc

在加锁的前提下，在进程表中找到一个未使用的进程，将其初始化：

- 用 `kalloc` 分配一个空页作为进程的内核栈
- 在内核栈上依次放置 trapframe, trapret, stack pointer 和 context
- 将 proc struct 的指针字段 tf 和 context 更新至上述栈上的位置
- 将 context->x30 即返回地址寄存器设为 forkret + 8 ： 作为新进程需要从 forkret 开始执行
- 设置 p->state 和 p->pid

user_init

在 `proc_alloc` 的基础上继续配置进程 p：

- 为其分配用户页表所需的一页内存
- 利用 `uvm_init` 将进程内存中的代码段装载
- 对 trapframe 中必要的部分进行配置：
  - 将 `spsr` 置0， 表示没有待处理的异常
  - 将 `sp` 设为用户栈的位置——页的地址最高处
  - 将 `elr` 设为0，即用户程序代码开始处，在 trapret 最后的 eret 指令会使 CPU 去执行
- 将 p->state 设为 RUNNABLE，使 `scheduler` 能够调度

uvm_init

分配一块新内存装载以 binary 为起点，大小为 sz 的二进制代码。再将这块内存在页表中映射到0为起点的虚拟地址中。

scheduler

不断遍历进程表找到状态为 RUNNABLE 的进程调度运行。在通过 `swtch` 切换进程之前需要设置当前 CPU 的进程标记 c->proc ，切换 ttbr0 为该进程的页表（对于第一个进程来说即 `user_init` 中配置的那个），将当前进程的 p->state 设为 RUNNING 防止别的 CPU 运行。

swtch

作为整个上下文切换机制中的关键函数，其功能反而相对简单。只需做两件事：

- 保存旧 context：把所有的 callee-saved registers 压入当前栈中，并把栈指针存入参数1指定的位置
- 载入新 context：把参数2指向的 context 寄存器悉数复原

最后跳转到新 context 的返回地址寄存器 x30 处。

forkret

x30 指向了这里，程序在此释放（`scheduler` 获取的） ptable 锁后落入到 `trapret` 中。

trapret

CPU 在此恢复 trapframe 的数据，由于之前已经将 `elr` 配置为0， eret 后将直接进入用户程序代码（虚拟地址0处开始）。

init

在我们的第一个用户进程中，分别通过 svc 进行了两个系统调用。

系统调用经过中断向量表和 `trap` 两次路由后进入 `syscall`，再经过判断参数 r0 即可调出相应的系统函数。