Lab5实验报告

问题一

trapframe 和 context 存在该进程的内核栈中。

context 存在进程栈栈顶的位置，这样 context 指针不仅可以表示该结构体，还可以用来表示进程栈的栈指针，方便上下文切换；trapframe 存在进程栈中，当进程出现异常，进入异常处理程序前保存进程运行状态时，ttbr1_el1 仍然是进程原先的，可以直接通过访问指针来的修改进程栈中的 trapframe 的值。

直接存在 proc 中也可以。需要把之后所有对 context 和 trapframe 的访问都调用 proc 中对应结构体的指针。另外，context 中需要添加一个值保存栈指针 sp。

问题二

void swtch(proca, procb) 的目的是保存 *proca* 的现场，并切换到 *procb* 。此时，内核栈使用的是 *proca* 的。保存现场意味着需要往 *proca* 的内核栈（当前使用的内核栈）中压入寄存器值，完成之后 *proca* 的 context 就存放在其内核栈栈顶的位置。这时就更新其 context 的指针指向内核栈栈顶位置。为了更新指针的值就需要传入指针的指针（ context** ）。如果使用 void swtch(struct context *, struct context *) 则需要在 context 中新添加一项保存栈指针 sp。

在程序行为中，caller-saved registers 是由调用者保存的。当发生切换时 caller-saved registers 已经保存在调用者自己的栈中了，在调用程序返回后，caller 会从自己的栈中把这些寄存器重新载入，不会使用 callee 返回时的值。所以无论是否保存 caller-saved regisgers，程序的运行都不会改变。

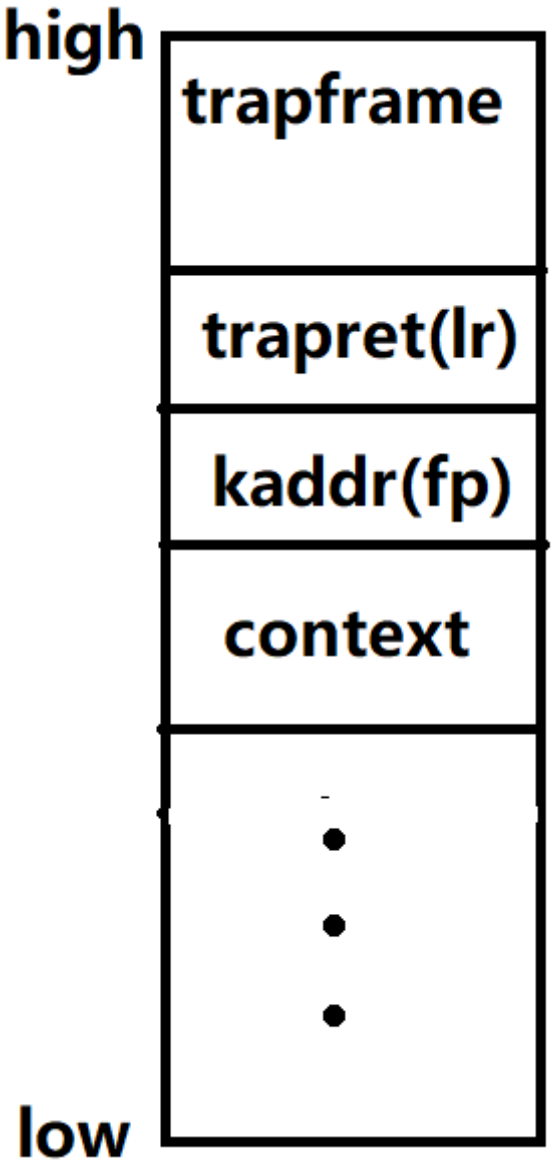
trapframe 需要把出现异常时的 pc、sp、pstate 和 caller-saved 寄存器保存下来。因为进程切换时 pc 寄存器会由 x30 寄存器保存，sp 寄存器会由 caller 在 proc 结构体中的 context** 保存，而 pstate 在没有发生异常的情况下值的都是不变的，但发生异常或中断时这些寄存器都可能会改动。另外，由于异常可能会在程序保存 caller-saved 寄存器出现（外部中断、时钟中断、内核栈溢出），这时 caller-saved 寄存器不知道保存了多少，所以 caller-saved 也需要在 trapfame 中。

上下文切换时需要进入到 scheduler 程序中进行，同时切换进程时涉及到换页表等操作，此时处理器应该在 el1 下以获得权限；而执行进程在 el0 下以防止进程得到高权限对系统造成破坏。所以需要调用 trap 切换到 el1 执行 scheduler ，之后再更新 context，最后 trapret 回到 el0 执行新进程。

程序说明

进程启动

如图是初始化时 trapframe 和 context 在栈中的存储方式。



在 trapframe 中，主要是把 spsr 初始化为程序入口
在 context 中，主要是把 x30（lr）、x29（fp）寄存器初始化为 trapret 入口。
在 trapframe 和 context 之间需要保存 x30、x29 寄存器的值，指向 forkret + 8 的位置。

启动进程时，scheduler 在切换完页表后会先调用 swtch 函数，完成上下文切换后 x30、x29 寄存器会从初始化栈中的 context 结构体读出。之后根据 x30 寄存器执行 b 指令。由于 x30 初始化为 forkret + 8 的值，会进入到汇编指令中相应位置，以跳过 forkret 汇编代码开头往栈中压入当前 x30、x29 的指令。完成 forkret 之后，会从栈中弹出 x30、x29 的值，由于之前没有压入，此时从栈中弹出的会是初始化栈时的在 context 之后的两个量 lr、fp，也就是 trapret 的位置。forkret 以 b x30 结束也就意味着进入 trapret。完成 trapret 后程序又用 eret 跳转到 spsr 寄存器指向的位置，也就是目标程序入口，同时切换回 el0，完成进程启动。

系统调用

当遇到中断、异常或进程调用 syscall 时，处理器会换到 el1 状态并进入 trap 函数。该函数会根据异常信息选择对应的处理入口并执行相应程序。处理完异常之后，trap 结束并通过 eret 重新返回到原进程。

具体程序（yield、exit等）在 xv6 手册中可查到，不详细讲述。

运行指令

make qemu