特权级转换：U 与 S 之间的切换

U 切换到 S：

- 当执行一个 trap 时，除了 timer interrupt，所有的过程都是相同的，硬件会自动完成下述过程：
  - 如果该 trap 是一个设备中断并且 sstatus 的 SIE bit 为 0，那么不再执行下述过程
  - 通过置零 SIE 禁用中断
  - 将 pc 拷贝到 sepc
  - 保存当前的特权级到 sstatus 的 SPP 字段
  - 将 scause 设置成 trap 的原因
  - 设置当前特权级为 supervisor
  - 拷贝 stvec（中断服务程序的首地址）到 pc
  - 开始执行中断服务程序

  CPU 不会自动切换到内核的页表，也不会切换到内核栈，也不会保存除了 pc 之外的寄存器的值，内核需要自行完成。其实切换页表的过程也很简单，只需要将内核的页表地址写入 satp 寄存器即可。

特权级转换：U 与 S 之间的切换

S 切换到 U：

- 在从 S 切换到 U 时，要手动清除 sstatus 的 SPP 字段，将其置为零；将 sstatus 的 SPIE 字段置为 1，启用用户中断；设置 sepc 为用户进程的 PC 值。如果启用了页表，就需要还原用户进程的页表，即将用户进程的页表地址写入 satp，之后恢复上下文，然后执行 sret 指令，硬件会自动完成以下操作：
  - 从 sepc 寄存器中取出要恢复的下一条指令地址，将其复制到程序计数器 pc 中，以恢复现场；
  - 从 sstatus 寄存器中取出用户模式的相关状态，包括中断使能位、虚拟存储模式等，以恢复用户模式的状态；
  - 将当前特权模式设置为用户模式，即取消特权模式，回到用户模式。

实验四：首个用户态进程创建

- 实验目标：在上一阶段的基础上，启动首个用户态进程proczero ，该进程执行两个简单的系统调用，随后进入死循环即可。即完成启动后，0号CPU陷在首进程用户态的死循环中，其余CPU陷在main()末尾的死循环中。
- 本阶段不实现：进程状态、多进程、时间片、有实际功能的系统调用、文件系统。

实验四：首个用户态进程创建

- 任务一：首个进程proczero的定义和初始化

1、进程定义

```
typedef struct proc {

    int pid;                    // 标识符
    pgtbl_t pgtbl;              // 用户态页表
    uint64 heap_top;            // 用户堆顶(以字节为单位)
    uint64 ustack_pages;        // 用户栈占用的页面数量
    trapframe_t* tf;            // 用户态内核态切换时的运行环境暂存空间

    uint64 kstack;              // 内核栈的虚拟地址
    context_t ctx;              // 内核态进程上下文

} proc_t;
```

实验四：首个用户态进程创建

■ 上下文与上下文切换

定义完进程后我们需要在 **cpu.h** 里新增两个字段:

- cpu 运行的进程 (用户进程)
- cpu 的上下文

在start.c中定义了CPU（ hart ）的栈：

__attribute__ ((aligned (16))) uint8 CPU_stack[PGSIZE * NCPU]

高级进程和低级进程都需要一个 **上下文 context_t**

- 高级进程每个CPU只有一个, 所以它的上下文保存在 cpu_t
- 低级进程可能很多, 所以上下文保存在 proc_t，**在proc.h中定义的。**

实验四：首个用户态进程创建

- **trapframe的定义（ proc.h ）**
  - trapframe的定义和 context有些类似, 看起来都是一堆寄存器信息。
  - 区别在于 context 是进程切换的暂存区, trapframe是同一个进程在内核态和用户态切换的暂存区。
  - trapframe 里的字段可以分成两部分去看:
    - 第一部分是: kernel_xxx 的内核信息 + epc 模式切换的返回地址
    - 第二部分是: 通用寄存器信息(比 `context` 更全, 模式间切换比同模式内切换需要保存的信息更多)

实验四：首个用户态进程创建

2、CPU进程创建proczero

proc.c 函数填写：

```
pgtbl_t  proc_pgtbl_init(uint64 trapframe);
void     proc_make_first();
```

在 **main.c** 的最后，CPU0 会调用 proc_make_first 创建第一个进程。

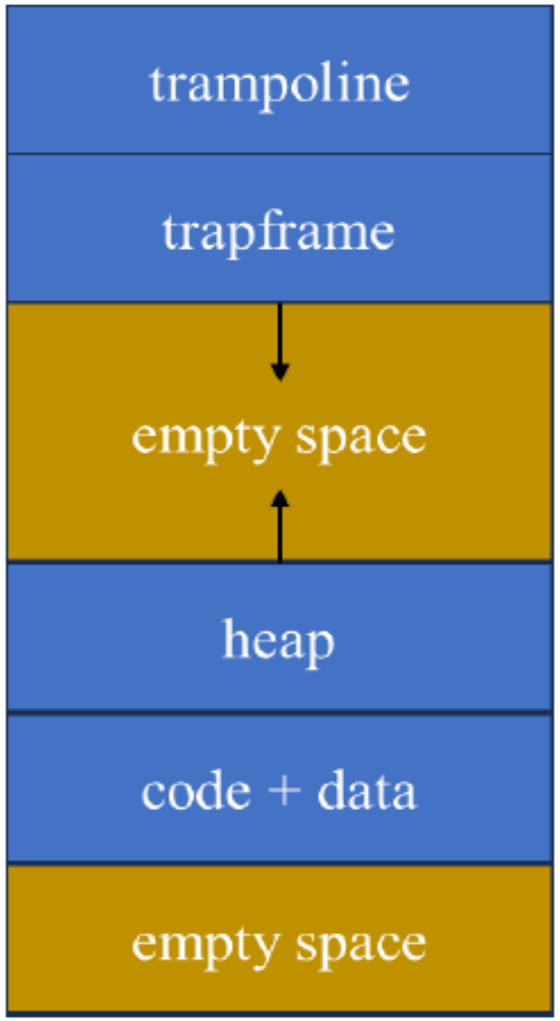
- proc_make_first 需要做的事情可以分成三部分：

（1）准备用户态页表 ：完成一系列映射以实现注释里的用户地址空间，需要调用 proc_pgtbl_init。

注意：需要在kvm_init里增加trampoline和kstack(需要pmem_alloc，使用KSTACK宏定义)的映射。在kernel.ld中增加trampoline页相关的内容，保证trampoline的程序是页对齐的。

实验四：首个用户态进程创建

- 跳板页（trampoline page）通常用于上下文切换过程。在上下文切换中，操作系统需要保存和恢复进程的状态。通过在内核和用户态之间共享跳板页，可以简化这种转换过程。
- 内核栈页：每个进程都有自己的内核栈。



实验四：首个用户态进程创建

（2）CPU进程将CPU使用权交给proczero

设置 **proczero** 的各个字段（trapframe里需要设置epc和sp, context里需要设置ra和sp）

（3）设置 CPU 当前运行的进程为 **proczero**, 使用 swtch切换进程上下文（CPU的ctx 到 proczero的ctx）

实验四：首个用户态进程创建

注意：

（1） 假设栈的地址空间是 [A, A+PGSIZE)，那么栈的指针初始应该设置为 A+PGSIZE 而不是 A（sp是向下运动的）

（2）关心 **initcode.h** 是如何在 **user** 目录中编译链接生成的，以及 **initcode.c** 做了什么事。

顺利执行的话，执行流会来到 trap_user_return，这就引出了我们的第二个任务：**用户态陷阱。**

实验四：首个用户态进程创建

- 首进程程序initcode.c

上述代码不能直接使用，老师已将其编译并导出为字节数组，将以上数组复制到proc.c的initcode[]数组中。

```
int main() {
        syscall(sys_print);   // 封装12号系统调用，a7寄存器装入12后执行ecall指令
        syscall(sys_print);   // 再次发起相同系统调用
        while(1) ;      // 死循环
        return 0;
}
```

```
uchar initcode[] = {
        0x13, 0x01, 0x01, 0xff, 0x23, 0x34, 0x81, 0x00, 0x13, 0x04, 0x01, 0x01,
        0x93, 0x08, 0x00, 0x00, 0x73, 0x00, 0x00, 0x00, 0x73, 0x00, 0x00, 0x00,
        0x6f, 0x00, 0x00, 0x00
};
```

实验四：首个用户态进程创建

- 任务二： 用户态陷阱处理

trap流程：

- 内核态陷阱处理流程是：kernel_vector前半部分->trap_kernel_handler->kernel_vector后半部分
- 用户态陷阱处理流程类似：user_vector->trap_user_handler->trap_user_return->user_return

很明显, 用户态trap处理被拆分成了上下两部分, 为什么?

原因是这样的处理流程假设用户进程一开始就是在用户态的, 但是事实上用户进程出生在内核态，于是初生的用户进程进入用户态的路径是: proc_make_first->trap_user_return->user_return-> U-mode

所以必须把trap处理函数分成两部分以兼容这条分支路径

实验四：首个用户态进程创建

trap实现

- trap的实现只需关注：**trampoline.S** 和 **trap_user.c**
- 这两个文件里共有四个函数, 分别是上一节里提到的用户态陷阱处理流程的四个阶段，汇编部分的代码已经写好且有完整注释, 需要大家阅读理解。
- trap_user_handler 的实现与 trap_kernel_handler 非常类似, 大部分代码可以直接copy，trap_user_return 的实现涉及很多寄存器和 **trapframe** 的操作, 确保你完全理解xv6的实现后完成。
- 另外, 在 **initcode.c** 里, **proczero** 向操作系统发出了第一个系统调用请求, 需要在 trap_user_handler 中进行响应。

响应方式很简单：printf("get a syscall from proc %d\n", myproc()->pid);

实验四：首个用户态进程创建

- 特别注意：全局数据区需要初始化为0！如果不做，会发生奇怪的错误。