指导书：

如何第一次从S态进入到U态?

需要在内核态触发一个异常，从而借助异常处理机制的返回流程进行上下文的切换，从而第一次进入到用户进程。

我们在 proc_init() 函数里初始化进程的时候, 认为启动时运行的ucore程序, 是一个内核进程("第0个"内核进程), 并将其初始化为 idleproc 进程。

lab5 新建了一个内核进程，执行函数 user_main() , 这个内核进程里我们将要开始执行用户进程。

我们在 user_main() 所做的，就是执行了 `kern_execv()` 这么一个函数。

实际上，就是加载了存储在这个位置的程序 exit 并在 user_main 这个进程里开始执行。这时 user_main 就从内核进程变成了用户进程。

在用户进程管理中，有几个关键的系统调用尤为重要：

• sys_fork() 用于创建当前进程的副本，生成子进程。父子进程都会从 sys_fork() 返回，但返回值不同：子进程得到0，父进程得到子进程的 PID ，这使得两个进程可以执行不同的代码路径。
• sys_exec() 在当前进程内启动一个新程序，保持 PID 不变但替换整个内存空间和执行代码。 fork() 和 exec() 的组合是 Unix-like 系统中创建新进程的经典方式。
• sys_exit() 用于终止当前进程，释放其占用的资源。
• sys_wait() 使当前进程挂起，等待特定条件（如子进程退出）满足后再继续执行。

在用户态进行系统调用的核心操作是，通过内联汇编进行 ecall 环境调用。这将产生一个 trap , 进入 S mode 进行异常处理。

trap.c 转发这个系统调用,trap.c略 具体函数名：
// kern/trap/trap.c
void exception_handler(struct trapframe *tf)

通过 kernel_execve 来启动第一个用户进程，进入用户态

如何实现 kernel_execve() 函数? 能否直接调用 do_execve() ?

-A:不能。do_execve() load_icode() 里面只是构建了用户程序运行的上下文，但是并没有完成切换。上下文切换实际上要借助中断处理的返回来完成。直接调用 do_execve() 是无法完成上下文切换的。如果是在用户态调用 exec() , 系统调用的 ecall 产生的中断返回时， 就可以完成上下文切换。但是，目前我们在 S mode 下，所以不能通过 ecall 来产生中断。

解决方式：这里采取一个取巧的办法，用 ebreak 产生断点中断进行处理，通过设置 a7 寄存器的值为10说明这不是一个普通的断点中断，而是要转发到 syscall() , 这样用一个不是特别优雅的方式，实现了在内核态复用系统调用的接口。

用户进程比起内核进程多了一个"用户栈"，也就是每个用户进程会有两个栈，一个内核栈一个用户栈，所以中断处理的代码 trapentry.S 要有一些小变化。

相关文件：# kern/trap/trapentry.S

进程退出：

ucore 分了两步来完成这个工作，首先由进程本身完成大部分资源的占用内存回收工作，然后由此进程的父进程完成剩余资源占用内存的回收工作。

在用户态的函数库中提供了 exit 函数，此函数最终访问 sys_exit 系统调用接口让操作系统来帮助当前进程执行退出过程中的部分资源回收。

exit 函数会把一个退出码 error_code 传递给 ucore，ucore 通过执行位于 /kern/process/proc.c 中的内核函数 do_exit 来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作

Exercise 1 附带问题：

load_icole函数讲解：

load_icode 是 ucore 内核的程序加载核心函数，负责把内存中的 ELF 格式应用程序（比如 ls、cat 这类可执行文件）“搬进” 当前进程的用户地址空间：
• 先清空进程旧的用户内存空间，搭建全新的内存管理骨架；
• 解析 ELF 程序的结构（代码段、数据段、BSS 段），把程序数据拷贝到物理内存并建立虚拟地址映射；
• 创建用户栈，配置进程的执行上下文（陷阱帧），最终让进程能从用户态执行新程序的第一条指令。(Exercise 1)

该函数被 do_execve 调用（对应 execve 系统调用

自己主要写的是第六部分：
`

```
// 核心设置：配置用户态执行的关键上下文
// (1) 设置用户栈指针（sp）：指向用户栈顶
tf->gpr.sp = USTACKTOP;
// (2) 设置用户态入口地址（epc）：ELF的入口地址（第一条指令地址
tf->epc = elf->e_entry;
// (3) 设置sstatus寄存器：配置用户态执行的权限和中断状态
tf->status = (sstatus & ~(SSTATUS_SPP | SSTATUS_SIE)) | SSTATUS_SPIE;
```

```
ret = 0;    // 核心设置：配置用户态执行的关键上下文
```

`

讲解：
trapframe（陷阱帧）是进程从内核态切换到用户态的指南，包含 CPU 执行用户态程序的关键上下文：
◦ tf->gpr.sp：用户态栈指针，指向栈顶；
◦ tf->epc：程序入口地址（ELF 的 e_entry），（执行第一条指令）”；
◦ tf->status：配置 sstatus 寄存器：
◦ 清除 SSTATUS_SPP：表示 “是用户态，不是内核态”；
◦ 清除 SSTATUS_SIE：内核态关闭中断，避免切换时被打断；
◦ 设置 SSTATUS_SPIE：用户态开启中断，允许响应外设（比如键盘、时钟）。

load_icole函数全程：

1. load_icode 的核心实现流程
2. 基础搭建：创建mm_struct和页目录表，为用户地址空间提供管理结构和页表基础；

3. ELF 解析：校验 ELF 合法性，遍历可加载段，创建 VMA 划分虚拟地址范围；
4. 内存加载：逐页分配物理内存，拷贝代码段 / 数据段，清零 BSS 段，构建完整的用户代码 / 数据空间；
5. 栈初始化：创建用户栈 VMA，预分配物理页，保证用户态栈可用；
6. 地址空间激活：关联mm到当前进程，加载页表到satp寄存器；
7. 上下文配置：清空旧陷阱帧，设置sp（用户栈顶）、epc（ELF 入口）、status（用户态特权级），保证sret能正确进入用户态执行第一条指令。

用户态进程从 RUNNING 到执行第一条指令的完整流程

当load_icode执行完成后，当前进程已具备完整的用户地址空间和上下文，接下来从 “ucore 调度器选中该进程（RUNNING 态）” 到 “执行应用程序第一条指令” 的全过程如下：

当 load_icode 执行完成后，进程已具备完整的用户地址空间和执行上下文，从 ucore 调度器选中该进程（RUNNING 态）到执行第一条指令的全过程如下（结合 ucore 内核机制）：
阶段 1：调度器选中进程，标记为 RUNNING 态

1. ucore 调度器（schedule 函数）遍历就绪队列，根据调度策略（如时间片轮转）选中该进程；
2. 调用 proc_run 函数触发进程上下文切换：
   ◦ 禁用中断（local_intr_save），保证切换过程原子性；
   ◦ 更新全局 current 指针为该进程，标记进程状态为 PROC_RUNNING；
   ◦ 加载该进程的页目录表物理地址到 satp 寄存器（lsatp(current->pgdir)），确保 CPU 使用该进程的页表解析虚拟地址。

阶段 2：内核态恢复进程上下文，准备切换到用户态

1. proc_run 调用 switch_to 函数，恢复该进程的内核上下文（context 结构体）：
   ◦ context 保存了进程上次在内核态暂停时的 CPU 寄存器状态（如 ra、sp 等）；
   ◦ switch_to 通过栈切换，将 CPU 寄存器恢复为该进程的内核上下文，跳转到该进程上次暂停的位置（即 do_execve 系统调用的返回路径）。

阶段 3：执行 sret 指令，完成内核态→用户态特权级切换

1. 进程在内核态执行到 do_execve 系统调用的返回逻辑，最终触发 sret 指令（RISC-V 异常返回指令）；
2. sret 指令读取 trapframe 中的关键字段，完成硬件上下文配置：
   ◦ 从 tf->status 加载 sstatus 寄存器：
   ◦ SPP=0：确认返回后进入用户态（U-mode）；
   ◦ SPIE=1：开启用户态中断；
   ◦ 从 tf->epc 加载 sepc 寄存器：将程序计数器（pc）设置为应用程序入口地址（elf->e_entry）；
   ◦ 从 tf->gpr.sp 加载用户态栈指针 sp：指向 USTACKTOP（用户栈顶）。

阶段 4：地址映射与执行第一条指令

1. CPU 切换到用户态后，解析 sepc 中的入口地址（elf->e_entry，用户虚拟地址）：
   ◦ 通过 satp 寄存器指向的页目录表，逐层查找页表项（PTE），将虚拟地址转换为物理地址（即 load_icode 中拷贝代码段的物理页）；
2. CPU 从该物理地址读取应用程序的第一条机器指令，完成指令译码、执行；

3. 用户态栈指针 sp 指向 USTACKTOP，保证程序启动时能正常使用栈（如执行 _start 函数、初始化 libc、调用 main 函数）。

关键： 调度器选中进程（RUNNING）→ proc_run 切换页表/上下文 → switch_to 恢复内核上下文 → sret 读取 trapframe 切换到用户态 → 地址映射 → 执行第一条指令

Exercise 2 附带问题：

Copy on Write (COW) 机制设计

概要设计：
fork 时，父子进程页表均指向同一物理页，并将页表项权限设为只读。每个物理页维护引用计数。

详细设计：

1. fork 时：
    - 父子页表均指向同一物理页，PTE_W 置零（只读），物理页引用计数+1。
2. 写入时：
    - 触发缺页异常，内核分配新物理页，拷贝原内容，更新页表为可写，原页引用计数-1。
    - 这样保证写操作互不影响，节省内存。

Exercise 3 问题：

1. fork/exec/wait/exit 执行流程（用户态 vs 内核态）
   （1）fork 执行流程
   • 用户态：进程调用 fork() 函数，触发 ecall 指令陷入内核；
   • 内核态：
2. 系统调用处理函数跳转到 do_fork；
3. 分配新 PCB、内核栈，复制父进程地址空间（copy_mm），设置子进程上下文；
4. 将子进程加入就绪队列，返回子进程 PID 给父进程，返回 0 给子进程；
   • 用户态：父子进程分别从 fork() 返回，继续执行后续代码；
   • 交互逻辑：用户态触发系统调用 → 内核态完成进程创建 → 内核态通过寄存器传递返回值 → 用户态恢复执行。

（2）exec 执行流程（对应 do_execve）
• 用户态：进程调用 execve(path, argv, envp)，触发 ecall 陷入内核；
• 内核态：

1. 校验入参，回收旧地址空间；
2. 调用 load_icode 加载新 ELF 程序，构建新用户地址空间；
3. 设置 trapframe 保证用户态执行新程序第一条指令；
4. 成功则无返回（程序已替换），失败则调用 do_exit；
   • 用户态：成功则执行新程序第一条指令，失败则进程退出；
   • 交互逻辑：用户态传递新程序信息 → 内核态完成地址空间替换 → 内核态配置执行上下文 → 用户态执行新程序。

（3）exit 执行流程
• 用户态：进程调用 exit(int code)，触发 ecall 陷入内核；
• 内核态：

1. 系统调用处理函数跳转到 do_exit；
2. 标记进程为 PF_EXITING，释放文件描述符、内存资源；
3. 设置退出码，将进程状态改为 PROC_ZOMBIE；
4. 唤醒父进程（若父进程在 wait），调用 schedule 让出 CPU；
   • 用户态：进程执行 exit 后不再返回，直接进入内核态处理；
   • 交互逻辑：用户态传递退出码 → 内核态完成资源预清理 → 内核态等待父进程回收最终资源。

（4）wait 执行流程（对应 do_wait）
• 用户态：父进程调用 wait(&status)，触发 ecall 陷入内核；
• 内核态：

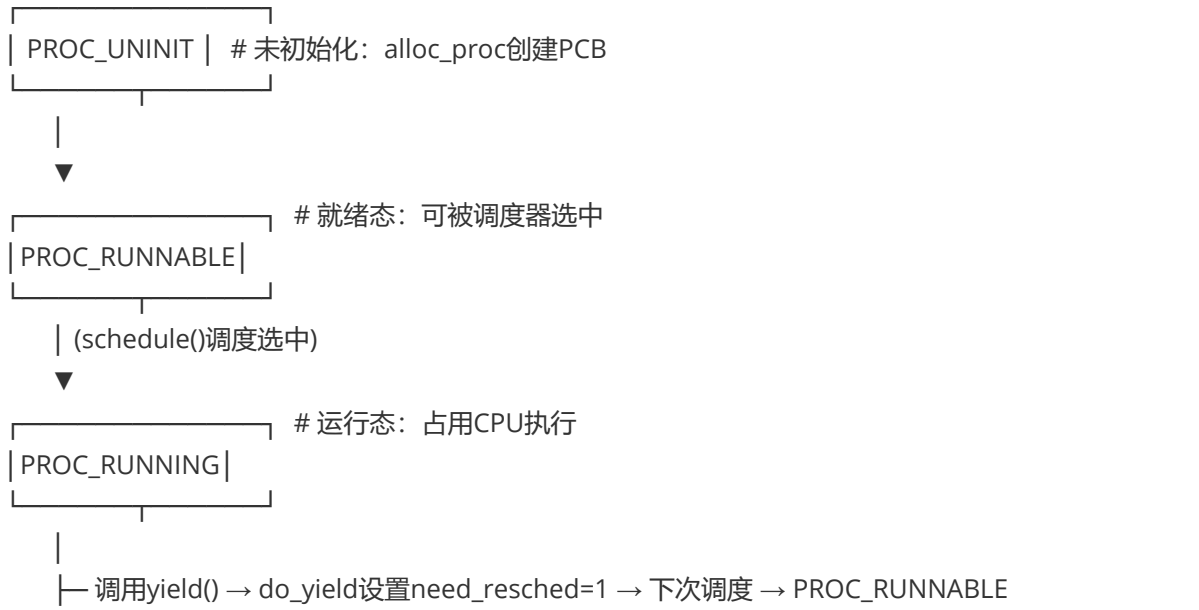
1. 校验入参，查找僵尸态子进程；
2. 无僵尸子进程则睡眠等待，被唤醒后重新查找；
3. 找到后将退出码写入用户态地址，回收子进程资源；
   • 用户态：父进程从 wait() 返回，获取子进程退出码，继续执行；
   • 交互逻辑：用户态传递退出码存储地址 → 内核态睡眠等待子进程 → 内核态写入退出码 → 用户态读取结果。

内核态与用户态交错执行 & 结果返回（重要！！！！！）
• 交错执行：（KEY!!!!!!!）

1. 用户态进程执行系统调用（fork/exec/wait/exit）时，ecall 指令触发异常，CPU 切换到内核态；
2. 内核态执行 do_xxx 函数完成核心逻辑（如创建进程、替换地址空间、等待子进程）；
3. 内核态通过 sret 指令返回用户态，进程从系统调用下一条指令继续执行；
4. 例：wait 调用时，父进程在 kernel 态的 do_wait 中调用 schedule 切换到其他进程；子进程退出后，父进程被唤醒，完成 do_wait 后返回用户态。

• 结果返回：

1. 简单返回值（如 fork 返回 PID、wait 返回 0）：内核态修改 CPU 寄存器（如 a0），sret 后用户态读取该寄存器；
2. 数据返回（如 wait 的退出码）：内核态直接写入用户态地址（需先校验合法性）；
3. 错误码：返回负数（如 -E_INVAL），用户态通过检查返回值判断出错类型。



YPTR OPTR CPTR

在 ucore 的 proc_struct 中，这三个指针用于构建父进程→子进程的链表，核心定义如下：

cptr　Child Pointer（子进程指针） 指向当前进程的第一个子进程
optr　Older sibling Pointer（兄进程指针） 指向同父进程的上一个兄弟进程（创建顺序更早）
yptr　Younger sibling Pointer（弟进程指针） 指向同父进程的下一个兄弟进程（创建顺序更晚）

三、正确的理解方式（用进程创建举例）
假设：进程 A（父进程）通过 fork 依次创建了进程 B、进程 C、进程 D（三个子进程），则三者的父子 / 兄弟指针关系如下：

1. 父进程 A 的指针
   A->cptr = B（A 的第一个子进程是 B）； A->optr/yptr 无意义（optr/yptr 仅对兄弟进程有效）。
2. 子进程 B 的指针
   B->parent = A（父进程是 A）； B->optr = NULL（B 是第一个子进程，无兄进程）； B->yptr = C（B 的下一个弟进程是 C）； B->cptr = NULL。
3. 子进程 C 的指针
   C->parent = A；C->optr = B（C 的上一个兄进程是 B）； C->yptr = D（C 的下一个弟进程是 D）。
4. 子进程 D 的指针
   D->parent = A；D->optr = C（D 的上一个兄进程是 C）； D->yptr = NULL（D 是最后一个子进程，无弟进程）。

父进程cptr → 第一个子进程 → 子进程yptr → 第二个子进程 → 子进程yptr → 第三个子进程 → NULL