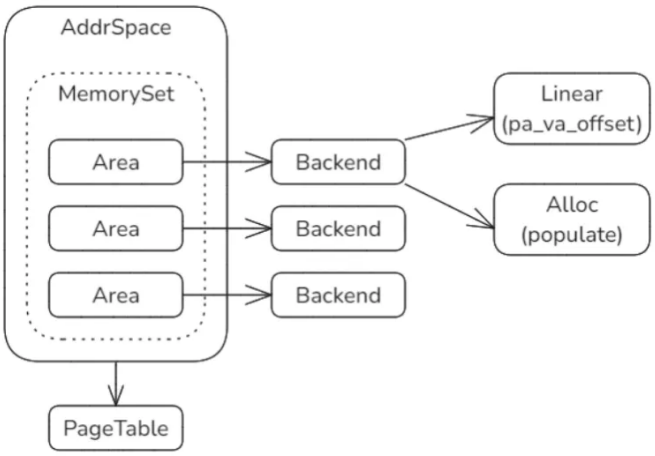
宏内核地址空间映射和Linux应用支持

M.2.0 UserAspace

ArceOS的地址空间管理结构

宏内核地址空间管理相关对象的层次构成

地址空间管理涉及的主要对象：AddrSpace，MemorySet，MemoryArea和Backend的两种实现。



AddrSpace：包含一系列有序的区域并对应一个页表。

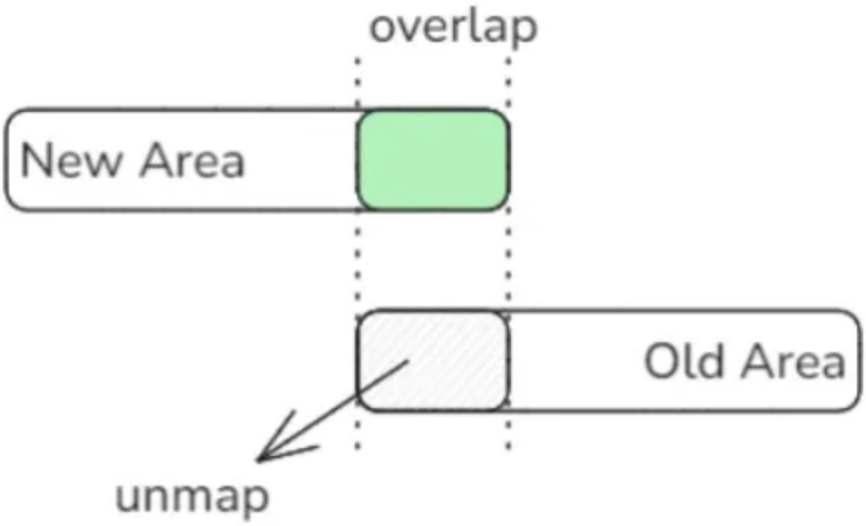
MemorySet：对BTreeMap的简单封装，对空间下的各个MemoryArea进行有序管理。

MemoryArea：对应一个连续的虚拟地址内存区域，关联一个负责具体映射操作的后端Backend。

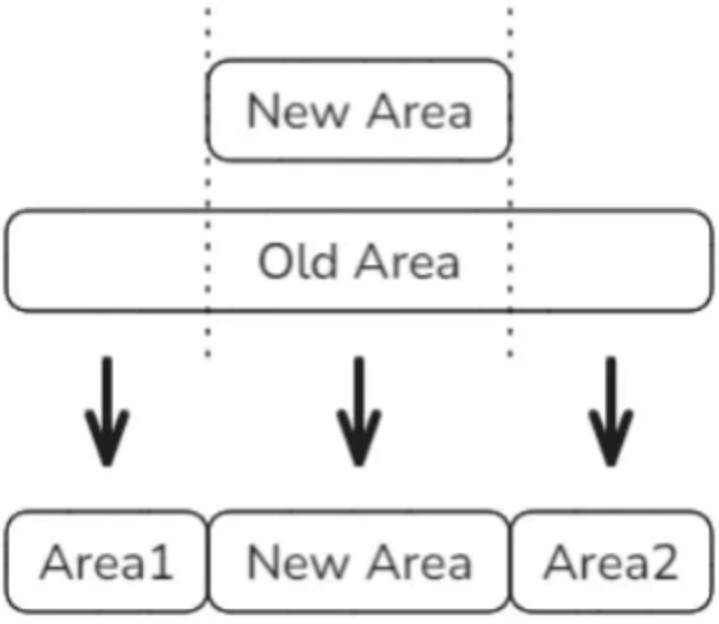
Backend：负责具体的映射操作，不同的区域MemoryArea可以对应不同的Backend。
目前支持两种后端类型：
Linear和Alloc。

地址空间重叠映射的策略

地址空间区域重叠时，原区域的重叠部分被Unmap，然后新区域被映射到空间中。



特殊情况：新区域位于旧区域的中间。



缺页异常处理和sys_mmap不同

实现方法类似，但是触发条件不同，缺页处理是中断响应，而mmap是由调用libc的某些方法实现

M.3.0 LinuxApp

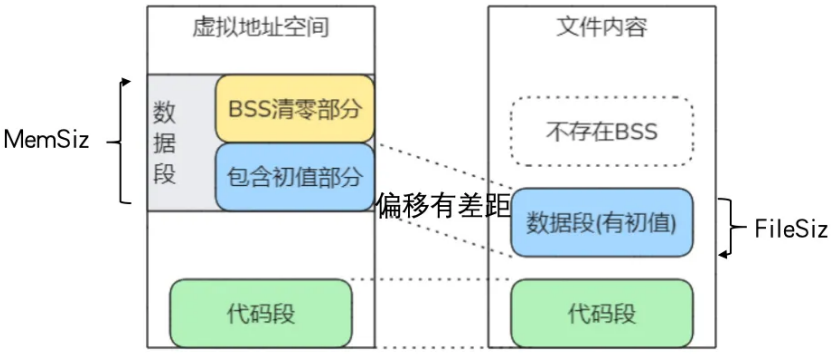
- 让Linux原始应用在宏内核上执行
- 需要兼容
  - Syscall
  - procfs & sysfs等伪文件系统
  - 应用、编译器和libc对地址空间的假定，涉及某些参数定义或某些特殊地址的引用

ELF格式应用的加载

- Entry point应用的入口地址
- 两个Type为LOAD的段，表示需要加载。分别是代码段和数据段，从Flag属性可以区分。注意：数据段的文件内偏移Offset和虚拟地址VirtAddr不一样，且FileSiz和MemSiz不一样

ELF格式应用的加载

需要注意文件内偏移和预定的虚拟内存空间内偏移可能不一致，特别是数据段部分。

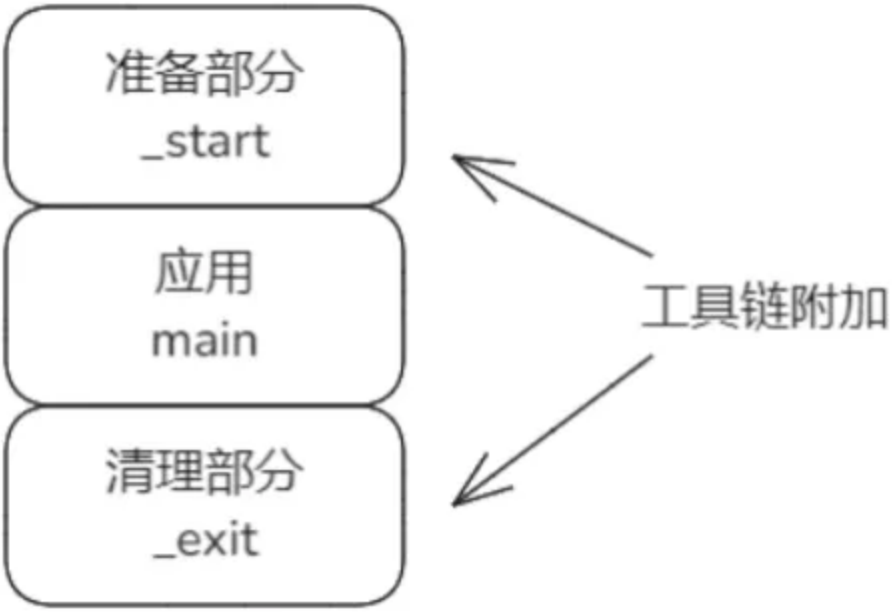


通常ELF为了节省空间，紧凑存储，从Offset和FileSiz定位段。
内核根据VirtAddr和MemSiz把段安置到目标虚拟内存位置。
由于BSS部分全零，所以ELF文件中只是标记位置和长度，不存实际数据，内核直接预留空间后**清零**。

应用的用户栈初始化

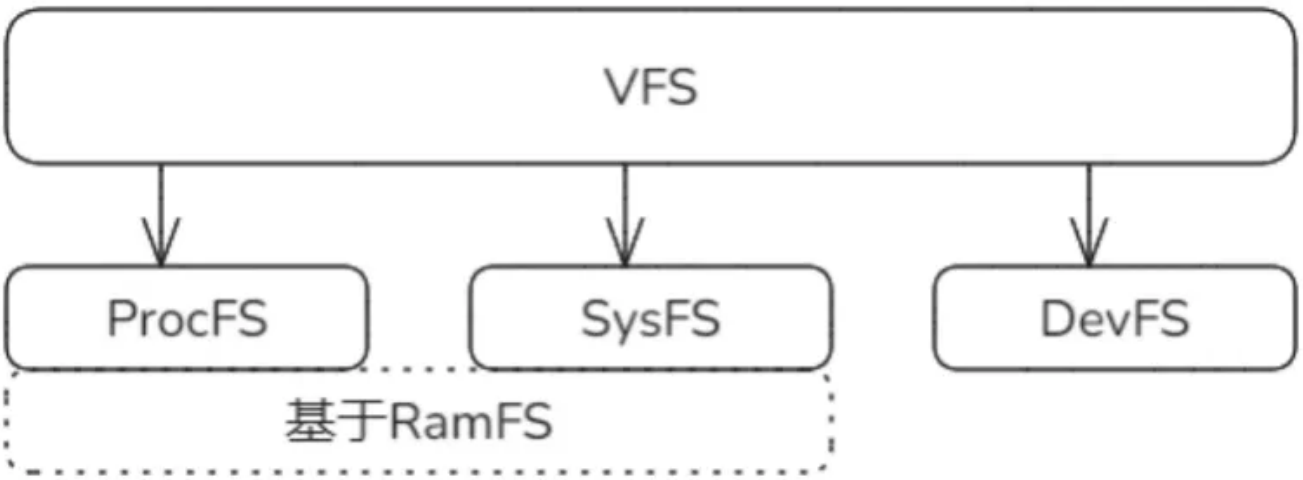
- Linux应用基于glibc/musl-libc等库编译，在执行main函数之前，要执行一些准备的程序，而

glibc和musl-libc使用的系统调用是不一样的，可能在写OS时会出现一些问题



Linux常用文件系统的支持

- Procfs：用于提供内核和进程信息的接口。它通常挂载在 /proc 目录下，包含了大量关于系统和进程的信息。
- Sysfs ： 用于向用户空间暴露设备信息。它通常挂载在 /sys 目录下。该文件系统主要用于替代传统的devfs。
- Devfs ： 用于向用户空间暴露设备和驱动信息。目前主要是为了兼容性而存在。



课后练习：实现mmap系统调用

遇到问题

- 一开始尝试没装musl-libc，尝试使用glibc，发现编译结束后直接pagefault，没有进入main函数，之后就装了musl-libc。

curl -O https://musl.cc/riscv64-linux-musl-cross.tgz

- 实际实现的话其实还好，主要就是要熟悉原来的内存分配的一些代码

```rust
fn sys_mmap(
    addr: *mut usize,
    length: usize,
    prot: i32,
    flags: i32,
    fd: i32,
    _offset: isize,
) -> isize {
    syscall_body!(sys_mmap, {
        let mut length = length;
        if (!length.is_aligned_4k()) {
            let padding = length - length % PAGE_SIZE_4K;
            length = padding + PAGE_SIZE_4K;
        }
        let curr = current();
        let curr_ext = curr.task_ext();
        let mut aspace = curr_ext.aspace.lock();
        let permission_flags = MmapProt::from_bits_truncate(prot);
        // TODO: check illegal flags for mmap
        // An example is the flags contained none of MAP_PRIVATE, MAP_SHARED, or MAP_SHARED_VALIDATE.
        let map_flags = MmapFlags::from_bits_truncate(flags);

        let start_addr = if map_flags.contains(MmapFlags::MAP_FIXED) {
            VirtAddr::from(addr as usize)
        } else {
            aspace
                .find_free_area(
                    VirtAddr::from(addr as usize),
                    length,
                    VirtAddrRange::new(aspace.base(), aspace.end()),
                )
                .or(aspace.find_free_area(
                    aspace.base(),
                    length,
                    VirtAddrRange::new(aspace.base(), aspace.end()),
```

问题 155 输出 终端 GITLENS

终端

```
MapFile ...
handle_syscall [56] ...
handle_syscall [64] ...
handle_syscall [57] ...
handle_syscall [56] ...
handle_syscall [66] ...
11111111
handle_syscall [222] ...
handle_syscall [66] ...
Read back content: hello, arceos!
handle_syscall [57] ...
handle_syscall [66] ...
MapFile ok!
handle_syscall [94] ...
[SYS_EXIT_GROUP]: system is exiting ..
monolithic kernel exit [Some(0)] normally!
```