Lab11 - Page Fault

0. 前言

实验概述

本节实验课将了解页错误机制，理解物理内存的按需分配，以及用户栈懒加载优化与mmap 文件映射功能。

实验代码

拉取实验代码：

```
$ git clone https://github.com/sdww0/sustech-os-lab.git && cd sustech-os-lab/lab11-pagefault
```

或更新实验代码：

```
# In sustech-os-lab
$ git pull && cd lab11-pagefault
```

本次实验新增：

1. **页错误处理器**：用于捕获并处理内存访问异常。
2. **VmArea**：表示一段连续的用户虚拟地址空间。它使用 `VmMapping` 记录单个页面的映射关系，并绑定了该区域特定的页错误处理策略。
3. **mmap 系统调用**：实现了以文件为后端的内存映射功能。
4. **性能测试程序**：新增 `fork_exec_time` 与 `fork_time` 用户程序，用于评估 Fork 与 Exec 的性能变化。

1. 页错误

页错误（Page Fault）是内存管理单元（MMU）在程序访问非法虚拟地址时触发的一种异常，通常发生在虚拟地址对应的物理页面不存在或当前访问权限不足时。例如，当用户程序尝试访问一个未映射的非法地址（如 0xdeadbeef）时，CPU 会捕获该异常并陷入内核态。如果该地址不属于进程的任何有效内存区域（如栈、堆等），内核通常会采取终止进程或发送信号（如 Linux 下的 SIGSEGV 段错误）等措施。

以下 C 程序演示了典型的非法地址访问：

```c
int main()
{
    int *crash_address = 0xdeadbeef;
    *crash_address = 42; // This will cause a segmentation fault
}
// [1]    4072580 segmentation fault (core dumped)  ./a.out
```

RISC-V 架构下的页错误处理

在 RISC-V 架构中，当发生页错误时，硬件会自动将相关信息保存到特定的控制与状态寄存器（CSR）中，供内核读取：

- `scause`：记录异常的具体原因，包括指令页错误、加载（Load）页错误或存储（Store/AMO）页错误。
- `stval`：记录导致触发页错误的具体虚拟地址。

内核页错误处理函数

我们在内核中实现了如下页错误处理入口函数：

```
pub fn page_fault_handler(
    process: &Arc<Process>,
    cpu_exception: &CpuExceptionInfo,
) -> core::result::Result<(), ()> {
    let memory_space = process.memory_space();
    let page_fault_addr = cpu_exception.page_fault_addr;

    let mut areas = memory_space.areas.lock();
    for area in areas.iter_mut() {
        if !area.contains_vaddr(page_fault_addr) {
            continue;
        }

        return area
            .handle_page_fault(process, page_fault_addr, cpu_exception.code)
            .map_err(|_| ());
    }

    Err(())
}
```

该函数的逻辑是：检查触发页错误的虚拟地址是否属于用户空间中某个已注册的 `VmArea`。如果是，则调用该区域特定的处理逻辑。`VmArea` 的结构定义如下：

```
/// Represents a continous virtual memory area, which consists of multiple mappings.
#[derive(Debug)]
pub struct VmArea {
    base_vaddr: Vaddr,
    pages: usize,
    perms: PageFlags,
    mappings: LinkedList<VmMapping>,
    fault_handler: Arc<dyn PageFaultHandler>,
}
```

VmArea 描述了一段连续的虚拟内存区域，包含起始地址、页面数量、权限标志、已建立的物理映射链表以及绑定的错误处理接口。对于错误处理函数，系统目前提供了两种主要实现：

1. `DefaultPageFaultHandler`：默认处理方式，对所有页错误均返回 Err()，表示非法访问。
2. `AllocationPageFaultHandler`：按需分配处理方式，当发生缺页时，动态分配一个新的物理页并建立映射。

页错误异常与内核功能

页错误异常不仅是错误处理机制，更是现代操作系统实现高级内存管理功能的基础。经典应用包括：

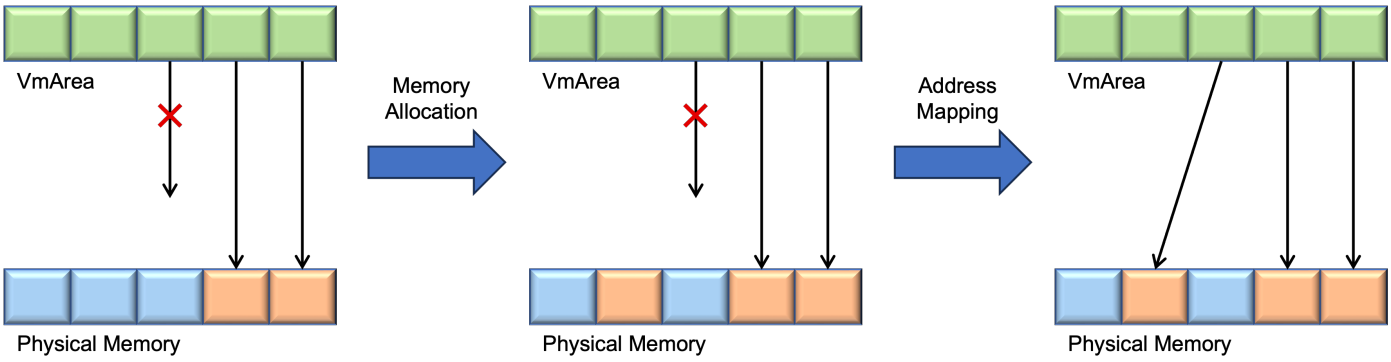
- **按需分配 (Demand Paging / Lazy Loading)**：当用户访问尚未映射的区域时，通过页错误陷入内核，内核再实时分配物理内存。这可以大幅减少内存浪费和启动时间。
- **交换技术 (SWAP)**：在物理内存不足时，将暂时不用的页面换出到磁盘，需要时再通过页错误换回，从而实现“无限”内存的假象。
- **写时复制 (Copy On Write, COW)**：在 fork 时不立即复制内存，而是将父子进程的页面映射设为只读。当任意一方尝试写入时触发页错误，内核才进行物理页的复制。这极大地优化了 fork 的性能。

本节课将重点介绍基于页错误实现的两个核心功能：（1）栈的按需分配；（2）mmap 内存映射。

2. 按需资源分配 - 栈

在之前的实验中，我们为每个进程的用户栈简单粗暴地预分配了固定大小的物理内存。然而，在实际场景中，绝大多数进程并不会用满整个栈空间，这种做法导致了显著的内存浪费。此外，执行 fork 系统调用时，必须完整拷贝整个栈的物理内存，严重拖累了进程创建的性能。

为了解决这些问题，我们可以利用页错误异常重构栈的实现，采用**按需分配**策略：初始时不分配物理页，当进程访问到尚未分配的栈地址时触发页错误，内核在处理程序中即时分配物理页面并建立映射。流程如下图所示：



Allocation Page Fault Handling

重构前后运行 `fork_exec_time` 与 `fork_time` 用户程序的性能对比：

- 重构前性能：

```
Hello World!, my pid: 100
Hello World!, my pid: 101
Hello World!, my pid: 102
Hello World!, my pid: 103

Result:
Total fork+exec count: 100
Total time: 4214000 microseconds
~ # fork_time
 Running command: fork_time
Starting fork performance test (100 forks)...

Result:
Total fork count: 100
Total time: 1826000 microseconds
~ #
```

- 重构后性能：

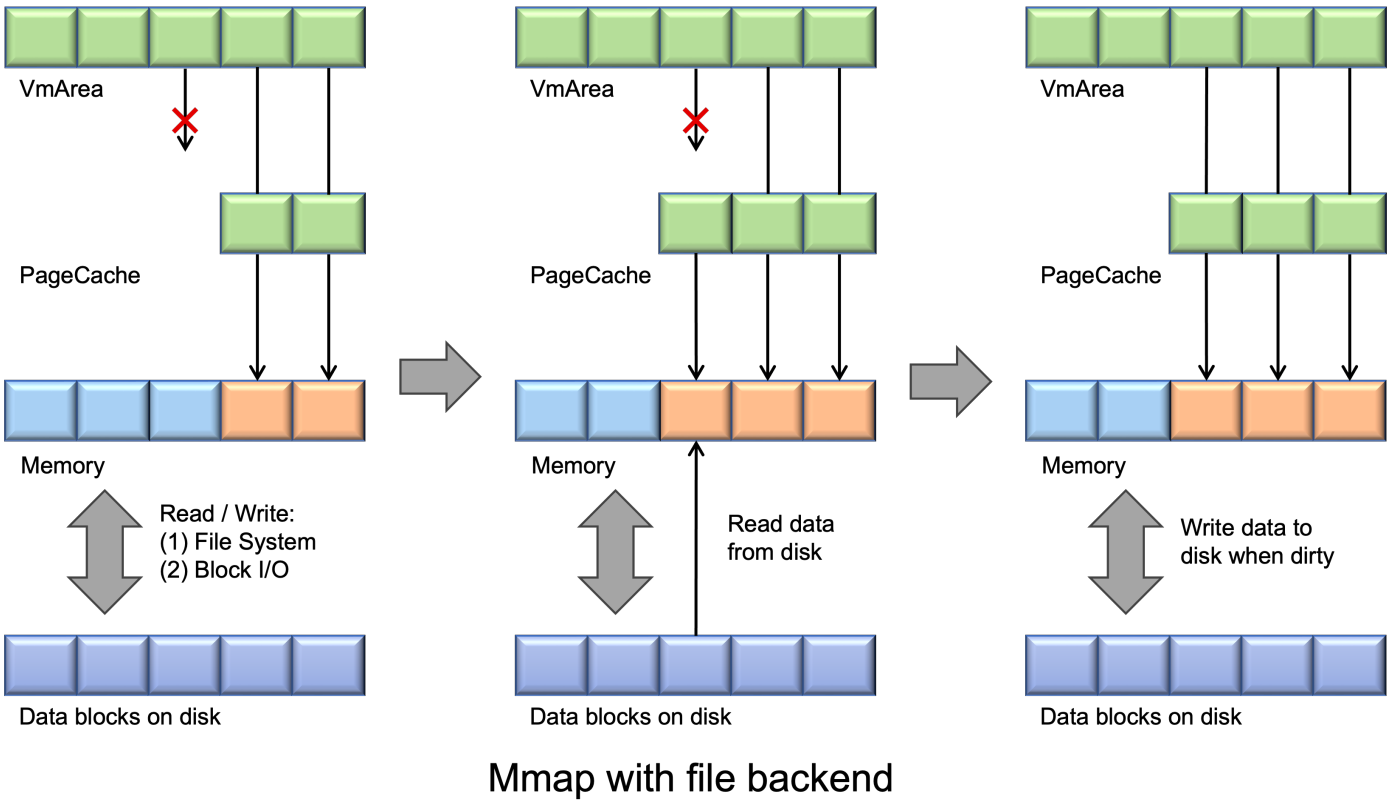
```
Hello World!, my pid: 101
Hello World!, my pid: 102
Hello World!, my pid: 103
Hello World!, my pid: 104

Result:
Total fork+exec count: 100
Total time: 1945000 microseconds
~ # fork_time
 Running command: fork_time
Starting fork performance test (100 forks)...

Result:
Total fork count: 100
Total time: 370000 microseconds
~ #
```

结果表明，优化效果显著：fork+exec 的组合操作性能提升了 1.17 倍，而单独的 fork 操作性能更是提升了 4.94 倍。

3. mmap内存映射

mmap（Memory Mapped）是一种高效的内存管理机制，它允许将文件或其他对象直接映射到进程的虚拟地址空间。通过 mmap，进程可以像访问普通内存数组一样读写文件内容，从而避免了频繁使用 read/write 等系统调用。mmap 不仅支持文件映射和匿名映射，还是实现高效进程间共享内存的基础。在本节实验中，我们将支持以文件作为后端的 mmap 映射（注：匿名映射的按需分配原理与前文所述的栈重构类似）。

以文件作为后端的 mmap 映射流程大致如下：



Mmap with file backend

在这个过程中，页错误处理往往需要与 **PageCache** 协同工作。

PageCache（页缓存）简介

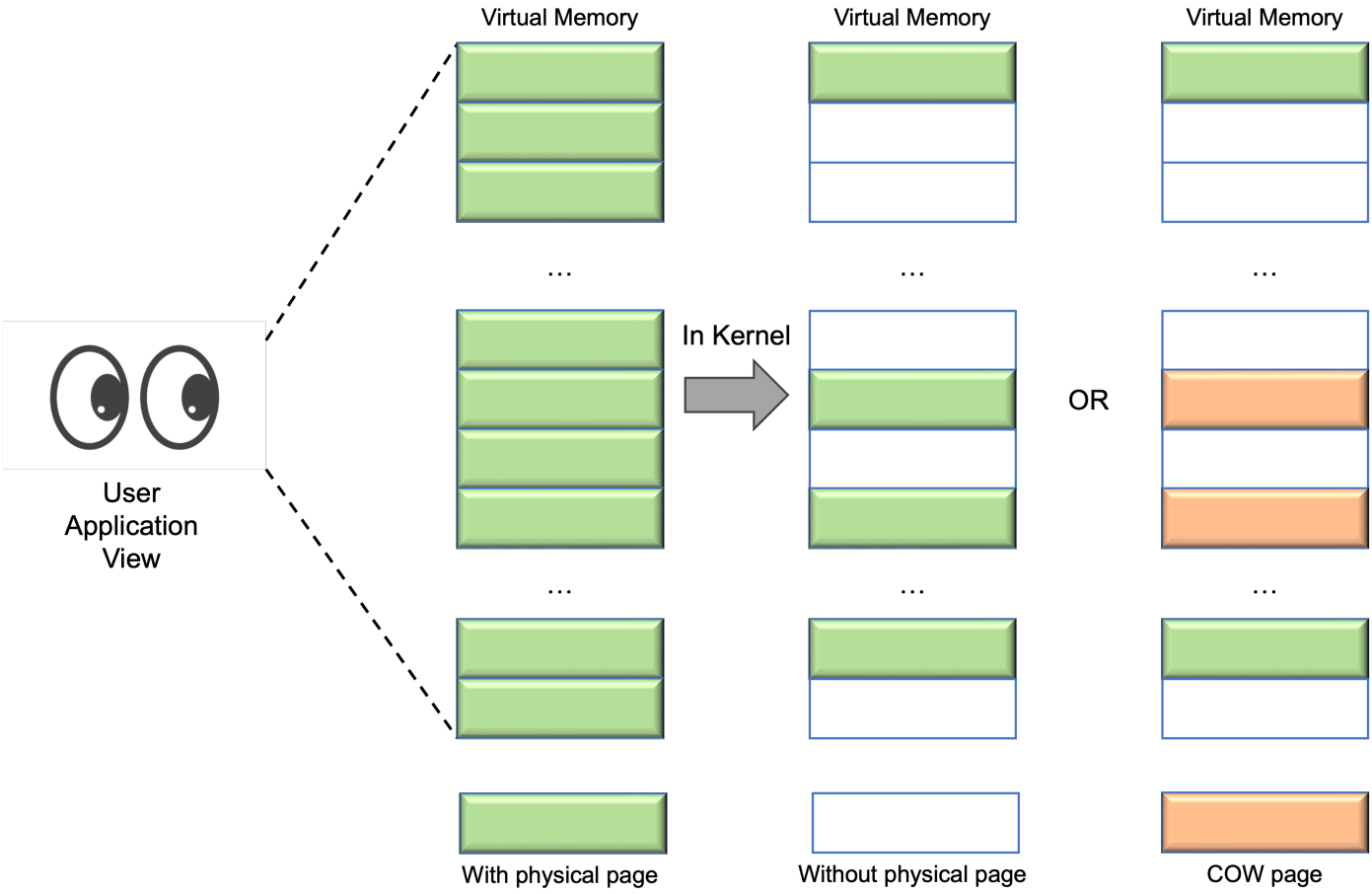
PageCache 是操作系统内核在内存中维护的一块缓存区域，用于存储从磁盘读取的文件页。

1. **缓存加速**：由于磁盘 I/O 的速度远低于内存，频繁的磁盘访问会成为系统瓶颈。PageCache 将访问过的文件数据缓存在内存中，当再次读取相同数据时，内核直接从内存返回，避免了缓慢的磁盘操作。
2. **预读与回写**：内核通常会预读数据到 PageCache 中以利用空间局部性；写入操作也可以先写入 PageCache（Write-back），稍后再异步刷入磁盘，从而极大提升 I/O 吞吐量。

在本次实验代码中，我们暂时未实现完整的 PageCache 结构，而是通过扩展 AllocationPageFaultHandler 实现了 MMapInodeFaultHandler。该处理函数在分配物理页后，增加了从 Inode 读取文件数据填充页面的步骤，以此模拟 PageCache 的基本行为。

4. 用户空间复杂化

引入页错误机制后，内核通过多种手段实现了性能加速与资源管理优化（如按需分配、COW 等）。随着内核功能的增强，用户空间的内存布局也变得更加复杂，如下图所示（其中COW将会在之后实现）：



在早期的实验代码中，用户空间的设计相对简单，主要由几块连续的、静态映射的虚拟地址到物理地址区域组成。而在引入高级功能后，内核中的用户空间不再是纯粹的连续已映射区域：

- 一个 VmArea 内部可能同时包含已映射的物理页和未映射的“空洞”。
- 某些页面可能处于 COW 状态（只读），等待写入时分裂；

这些复杂的底层状态通过页错误处理函数被封装在内核中，用户程序通常可以在无感知的情况下享受性能提升。

5. 上手练习

按需分配扩展 - 匿名映射

在此前的实验中，我们已基于按需分配策略重构了用户栈的实现，但目前的 mmap 系统调用尚未支持匿名映射。请你扩展现有的 mmap 实现，使其支持 MAP_ANONYMOUS 标志。相关细节请参考 mmap手册 中关于`The flags argument - MAP_ANONYMOUS`的说明。

完成后，请运行 mmap_anon_test 用户程序以验证你的代码。若实现正确，该程序的 fork 操作性能应该极快，且不会出现内存耗尽的现象。