内存映射（mmap）与 I/O 优化实验指导

一、本节内容导读（知识点）

本节围绕内存映射（mmap）机制 展开。mmap 是操作系统中连接虚拟内存管理与文件系统的关键技术，它通过页表将磁盘文件直接纳入进程的虚拟地址空间，使程序可以像访问内存一样访问文件数据。本节重点理解以下核心知识点：

---

1. 虚拟地址空间的进一步扩展方式

在传统的进程内存模型中，进程的虚拟地址空间主要由以下部分组成：

- 代码段（Text）
- 数据段（Data / BSS）
- 堆（Heap）
- 栈（Stack）

这些区域主要服务于**程序本身的数据和代码需求。** 而 **mmap 引入了一种新的地址空间扩展方式**：

> **将外部对象（文件、匿名内存）直接映射进进程的虚拟地址空间**

当调用 `mmap()` 时：

- 操作系统在进程的虚拟地址空间中划出一段连续区间；
- 该区间并不立即分配物理内存；
- 页表项被标记为“尚未映射到物理页”。

这意味着：

- 虚拟地址空间的大小可以远远超过物理内存；
- 映射区本质上是一种“惰性分配”的地址空间扩展；
- 实际物理页只有在访问时才会被分配或调入。

因此，mmap 是虚拟内存“按需使用”思想的典型体现。

---

2. 文件与内存的统一抽象

在不使用 mmap 的情况下，文件访问通常采用如下模型：

```
磁盘文件 → 内核缓冲区 → 用户缓冲区
```

该模型存在的问题包括：

- 需要显式调用 `read()` / `write()`；
- 存在用户态与内核态之间的数据拷贝；
- 随机访问需要频繁维护文件偏移量。

mmap 改变了这一模型：

```
磁盘文件 ⇔ 虚拟地址空间 ⇔ CPU 访问
```

通过 mmap：

- 文件被**抽象为一段连续的虚拟内存区域**；
- 程序可以使用普通的指针和数组操作访问文件内容；
- 不再区分“文件访问”与“内存访问”的代码逻辑。

从程序员视角来看：

> **文件不再是 I/O 对象，而是内存对象**

这种统一抽象大幅降低了程序复杂度，也为高性能 I/O 提供了基础。

---

3. 按需调页（Demand Paging）机制

mmap 映射文件时，并不会一次性将整个文件加载到内存中。 其背后依赖的是 **按需调页机制**：

1. 调用 `mmap()` 时：
    - 仅建立虚拟地址到文件的映射关系；
    - 不分配物理页、不进行磁盘读取。
2. 程序第一次访问映射区中的某个地址：
    - CPU 发现该页无效，触发缺页中断；
    - 内核根据页表信息定位对应的文件偏移；
    - 将文件内容读入物理页；
    - 更新页表，恢复进程执行。
3. 后续访问同一页：
    - 直接命中内存，不再触发 I/O。

这种机制的意义在于：

- 即使文件远大于物理内存，也可以被映射；
- 只加载“真正被访问”的页面；
- 显著降低内存占用和 I/O 压力。

这正是 mmap 能够高效处理大文件的根本原因。

---

4. 写时复制（Copy-On-Write, COW）

在使用 `MAP_PRIVATE` 方式映射文件时，mmap 引入了 **写时复制机制**：

- 初始状态下：

    - 映射区中的页面是**只读共享的**；
    - 多个进程可以映射同一文件而不发生冲突。

- 当进程尝试写入映射区：

    - CPU 触发写保护异常；
    - 内核为该进程分配新的物理页；
    - 将原页面内容复制到新页；
    - 页表指向新页，写操作在私有副本上完成。

结果是：

- 读操作始终共享；
- 写操作彼此隔离；
- 原始文件内容不会被修改。

COW 实现了：

> **内存共享与进程隔离之间的平衡**

它是 mmap、fork 等机制高效且安全运行的重要基础。

---

5. 页表权限位与内存访问安全

mmap 映射区域的访问权限由页表中的权限位控制，例如：

- `PROT_READ`：可读
- `PROT_WRITE`：可写
- `PROT_EXEC`：可执行

这些权限由硬件（MMU）直接强制执行：

- 非法访问不会被程序忽略；
- 会触发硬件异常（如 Segmentation Fault）；
- 内核据此终止进程或采取其他措施。

例如：

```
mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
```

此时若执行：

```
start[0] = 'A';
```

将触发写保护异常，进程被终止。

这表明：

- 内存安全不是“约定”，而是**硬件级别的强制约束**；
- mmap 在提供高效访问的同时，仍然保证了严格的安全隔离；
- 页表权限是操作系统实现安全性的重要基础设施。

通过 mmap 机制，操作系统实现了：

- 虚拟地址空间的弹性扩展；
- 文件与内存的统一访问模型；
- 按需加载突破物理内存限制；
- 写时复制保障共享与隔离并存；
- 硬件级权限控制确保内存安全。

这使得 mmap 成为现代操作系统中连接**存储管理、进程管理与 I/O 系统**的核心机制之一，也是理解虚拟内存设计思想的关键切入点。

二、实验目标

1. 理解 mmap 的工作原理及其与 read/write 的区别；
2. 掌握基于 mmap 的文件访问方式；
3. 理解按需调页如何突破物理内存限制；
4. 验证页表权限在内存安全中的作用；
5. 体会 mmap 对程序结构与 I/O 性能的简化效果。

三、实验内容

任务一：基于 mmap 的文件读取

传统的 read 需要将数据从磁盘拷贝到内核缓冲区，再拷贝到用户空间。而 mmap 利用页表将文件直接映射到进程的虚拟地址，实现了数据的直接访问。

```
struct stat sb;
fstat(fd, &sb); // 1. 动态获取文件大小，解决“地址分配复杂”问题

// 2. 建立内存映射：文件 --> 虚拟地址空间
start = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);

if (start != MAP_FAILED) {
    printf("%s\n", (char *)start); // 3. 像操作数组一样直接读文件
}
```

【代码详解】：

- 文件被映射为进程地址空间的一部分。
- fstat 动态获取：不需要手动硬编码地址或大小，系统自动根据文件长度分配虚拟空间。
- 写时复制（COW）：MAP_PRIVATE 确保了即使我们修改映射区，也不会破坏原始文件。这是导读中提到的“内存共享与隔离的平衡”的体现。

任务二：利用 mmap 实现分页阅读工具

步骤 1：映射与初始化

在传统 I/O 中，实现翻页需要频繁调用 lseek 或维护复杂的缓冲区。而使用 mmap，我们只需要操作一个内存指针即可完成复杂的随机跳转。

步骤 1：文件映射与初始化

首先获取文件大小并建立映射，将整个文件调入虚拟地址空间。

```
// 1. 获取文件大小以确定地址空间范围
struct stat sb;
fstat(fd, &sb);

// 2. 建立内存映射：整个文件映射为只读的字符数组
char *start = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
close(fd); // 映射成功后，文件描述符即可关闭，不影响映射区访问

// 3. 定义分页索引：存储每一页起始位置的偏移量，用于“回翻”
size_t page_starts[10000];
size_t pos = 0;         // 当前读取指针（相对于映射区首地址）
int current_page = 0;   // 当前页码
page_starts[0] = 0;     // 第 0 页从文件开头开始
```

【代码详解】：

- fstat 与 sb.st_size：确保 mmap 映射的空间完美覆盖文件内容，实现了地址分配的精细化。
- page_starts 数组：由于每行长度不等，我们必须记录每一页在映射区中的起点偏移量。这不仅解决了“回翻”难题，更体现了如何通过简单的数据结构辅助内存管理。

步骤 2：分页读取逻辑

通过扫描内存中的换行符 \n 来动态定义一页的边界。

```
while (pos < sb.st_size) {
    int line_count = 0;
    // 内层循环：输出一页内容（20 行）
    while (pos < sb.st_size && line_count < LINES_PER_PAGE) {
        putchar(start[pos]); // 直接读取内存内容
        if (start[pos] == '\n') {
            line_count++; // 扫描到换行符，行计数加 1
        }
        pos++;
    }
    // ... 等待用户翻页指令 ...
}
```

【代码详解】：

- 内存指针操作：无需 read() 即可访问数据。start[pos] 会自动触发操作系统的缺页中断，由内核按需将磁盘块载入，验证了导读中"动态调度与高效利用"的逻辑。
- 逻辑简化：分页逻辑被简化为了对内存数组的遍历，极大降低了维护文件指针的复杂度。

步骤 3：交互与回读逻辑（向前/向后导航）

mmap 的优势在于支持极高效率的非线性访问。

```
if (c == 'b') { // 用户请求返回上一页
    if (current_page > 0) {
        current_page--;
        pos = page_starts[current_page]; // 1. 将指针重置回上一页记录点
    }
    continue; // 2. 重新进入输出循环
}
// 用户按回车：进入下一页
current_page++;
page_starts[current_page] = pos; // 3. 记录新页面的起始偏移量
```

【代码详解】：

- 高效随机访问：回读不再需要 lseek 定位和重复 I/O。只需重置内存下标 pos 即可。
- 性能跨越：所有的翻页操作仅涉及虚拟指针的移动，不涉及用户态与内核态的数据搬运，解决了导读中的"I/O 性能瓶颈"。

任务三：关键原理解析：为什么 mmap 适合分页工具？

1. 统一编址：将磁盘文件抽象为地址空间的一部分，程序处理逻辑从"文件处理"简化为"内存处理"。
2. 按需加载（Demand Paging）：即使文件大小远超物理内存，系统也只需加载当前阅读的页面（20行内容所在的页），完美解决了导读中提到的"突破硬件限制"的问题。
3. 安全隔离：通过设置 PROT_READ，内核在硬件层面阻止了对文件的非法修改，实现了导读中要求的"安全隔离"。
4.

四、观察与思考

1. 为什么 mmap 后可以关闭文件描述符？
2. 如果文件很大但只读前几页，内存是否会暴涨？
3. 写入 `start[0] = 'A';` 会发生什么？
4. mmap 与 read 在性能与编程模型上的根本差异是什么？

五、小结

本实验展示了 mmap 如何将 I/O 问题转化为内存访问问题：

- 减少用户态/内核态切换；
- 支持高效随机访问；
- 利用虚拟内存机制实现性能与安全的平衡。
