动态内存分配与内存映射实验指导

一、本节内容导读（知识点）

本节围绕 **操作系统中的动态内存分配机制** 展开。动态内存分配是操作系统内存管理中最基础、也是最核心的功能之一，它直接决定了系统在多进程并发、内存资源受限的情况下，能否高效、稳定且安全地运行。本节重点理解以下几个方面的原理。

---

1. 物理内存与虚拟内存的区别与联系

物理内存是计算机中真实存在的硬件资源，即实际的 RAM 芯片，其容量有限、地址连续性受硬件约束。

虚拟内存则是操作系统向每个进程提供的一种抽象：

- 为每个进程构造一个**独立、连续、私有**的地址空间；
- 虚拟地址并不直接对应物理地址；
- 通过页表由硬件（MMU）完成虚拟地址到物理地址的转换。

二者之间的关系可以概括为：

> **虚拟内存是对物理内存的抽象、复用与扩展**

动态内存分配发生在虚拟地址空间中，而最终是否占用物理内存，则由操作系统在运行时根据访问行为动态决定。这种分离使得：

- 程序无需关心物理内存布局；
- 多个进程可以安全共享有限的物理内存；
- 系统可以通过按需分配和页面置换突破物理内存容量限制。

---

2. Linux 系统中内存使用状态的多维度表达方式

在 Linux 中，内存并非简单地分为“已用”和“未用”。为了提高整体性能，内核会尽可能利用空闲内存作为缓存，因此引入了多种状态指标。

典型维度包括：

- **Free**：完全未被使用、且未被任何对象占用的物理内存；
- **Cached / Buffers**：用于文件缓存和块设备缓存的内存；
- **Available**：在不影响系统运行的情况下，可以立即分配给进程使用的内存估计值。

这些指标共同反映了一个事实：

> **“被使用的内存”不等于“不可用的内存”**

动态内存分配依赖这种策略，使系统在保证应用可用性的同时，最大化物理内存利用率。理解这些指标，是正确判断系统内存压力和分配行为的前提。

---

3. `/proc` 文件系统在内存监控中的作用

`/proc` 是 Linux 提供的一种**虚拟文件系统**，用于向用户空间暴露内核内部状态。

在内存管理方面：

- `/proc/meminfo` 提供系统级内存统计信息；
- `/proc/[PID]/maps` 展示进程虚拟地址空间的布局；
- `/proc/[PID]/status` 提供进程的内存使用概览。

这些文件并不真实存在于磁盘上，而是由内核在读取时动态生成。 它们的作用在于：

- 让用户和调试工具"可视化"内核内存管理状态；
- 为性能分析和故障诊断提供依据；
- 将复杂的内核数据结构以统一接口呈现出来。

通过 `/proc`，动态内存分配从"黑箱"变成了**可观测、可分析的过程。**

---

4. 进程虚拟地址空间与内存隔离机制

每个进程都拥有独立的虚拟地址空间，通常包含：

- 代码段（只读、可执行）
- 数据段（读写）
- 堆（动态分配区）
- 栈（函数调用区）
- 映射区（共享库、mmap 区域）

内存隔离通过以下机制实现：

- **独立页表**：不同进程的虚拟地址可映射到不同的物理页；
- **权限位控制**：页表中记录读 / 写 / 执行权限；
- **硬件强制执行**：非法访问由 MMU 直接触发异常。

这意味着：

- 一个进程无法访问或破坏另一个进程的内存；
- 进程也无法随意访问内核空间；
- 动态分配的内存同样受到严格的隔离保护。

内存隔离是动态内存分配安全性的基础保障。

---

5. 内核级动态内存分配的基本思想（以空闲链表为例）

在内核态，无法直接使用用户态的 `malloc`。内核需要自行管理物理内存，常见方式是以**页为单位进行分配。**

空闲链表是最基础的管理策略之一，其核心思想包括：

- 将空闲物理页组织成链表；
- 利用空闲页自身的空间存储链表指针；
- 分配时从链表中取出一个页；
- 回收时将页重新挂回链表。

这种设计体现了内核动态内存分配的特点：

- **低开销**：不引入额外的数据结构存储；
- **高可靠性**：算法简单，行为可预测；
- **面向底层资源管理**：直接操作物理页，而非字节级内存。

尽管实际内核中还会使用更复杂的分配器（如伙伴系统、slab 分配器），但空闲链表清晰地展示了动态内存管理的基本思想。

通过对以上知识点的理解，可以建立如下认知：

- 动态内存分配运行在虚拟内存抽象之上；
- Linux 通过多维度内存状态提高资源利用率；
- `/proc` 提供了观察内存管理行为的窗口；
- 虚拟地址空间与权限位保障了安全隔离；
- 内核通过高效的数据结构管理有限的物理内存。

这些机制共同构成了操作系统在复杂运行环境中，实现**高效、稳定与安全内存管理**的基础。

二、实验目标

1. 掌握 Linux 系统与进程内存状态的查看方法；
2. 理解 Free、Available、Cache 等内存指标的真实含义；
3. 理解进程虚拟地址空间的布局及其安全隔离机制；
4. 掌握内核级动态内存分配（kalloc）的基本实现思路；
5. 建立从“系统监控”到“内核实现”的完整认知链路。

三、实验内容

任务一：系统与进程内存状态观测

1. 系统级内存观测

在 Linux 中，系统内存并不是简单的“已用”和“未用”，还包含缓存（Cache）和缓冲区（Buffer）。

```
# 查看系统实时内存快照
free -h
# 获取内核层面的精细统计
cat /proc/meminfo
```

【关键信息解读】：

- Available vs Free：Free 是完全未被使用的物理内存，而 Available 包含了可以被立即回收的缓存。在内存受限时，内核会动态释放缓存给应用使用。
- Committed_AS：这是导读中提到的“虚实映射”关键指标，代表当前系统所有进程申请的虚拟内存总量，即使物理内存不足，系统也能通过虚拟化技术提供超出硬件限制的地址空间。

> 说明：即使物理内存不足，系统也可以通过虚拟内存机制为进程提供更大的地址空间。

---

2. 进程级内存布局观测

通过查看进程的地址映射表，我们可以直观看到导读中提到的“虚实地址映射”和“内存隔离”。

```
# 查看特定进程（如 xeyes）的地址空间布局
cat /proc/[PID]/maps
```

【观察与思考】：

- 权限控制与安全隔离：注意每行中的 r-xp（代码段：只读、可执行）和 rw-p（数据段：读写、私有）。页表权限位确保了应用无法篡改自己的代码，也无法访问内核空间，实现了安全隔离。
- 地址空间简化：编译器在生成程序时，通常使用一致的虚拟起始地址。通过页表的虚实转换，操作系统让每个应用都觉得自己在独占整个内存空间。

任务二：内核级动态内存分配原理（kalloc）

在内核中，由于没有现成的堆，我们需要手动管理空闲物理页。

```
// 定义空闲链表节点，直接存放在空闲页的首地址处
struct linklist {
    struct linklist *next;
};

// 分配一个 4KB 的物理页
void *kalloc(void) {
    struct linklist *l;
    l = kmem.freelist; // 1. 指向当前链表头
    if (l) {
        kmem.freelist = l->next; // 2. 移除头部，更新可用列表
        memset((char *)l, 5, PGSIZE); // 3. 填充垃圾数据，防止残留引用
    }
    return (void *)l;
}
```

【代码详解】：

- 空间复用：struct linklist 本身不占额外内存，它直接利用空闲页面前 8 个字节存储指针。
- 分配算法：这里采用的是最简单的连续分配管理，通过链表动态维护一系列空闲块。

实现思想说明：

- 使用空闲物理页自身存储链表指针，实现空间复用；
- 每次分配从空闲链表头部取出一个页；
- 填充垃圾数据防止悬空引用。

四、观察与思考

1. 为什么系统中 Free 很小但程序仍能正常运行?
2. Available 内存的存在解决了什么问题?
3. 进程地址空间看似连续，实际上物理内存是否连续?
4. 内核级 kalloc 与用户态 malloc 在设计目标上有何不同?

五、小结

通过本实验，我们从系统视角与内核实现视角理解了动态内存分配机制：

1. Linux 通过多层指标精细描述内存状态；
2. 虚拟内存机制屏蔽了物理内存的复杂性；
3. 内核通过简单而高效的数据结构管理物理页；
4. 动态内存分配是操作系统稳定运行的基础能力之一。