# 物理内存管理及最简单的页表映射

## 操作系统 lab2 实验报告

梁景铭　王一诺　强博
南开大学计算机学院

2025 年 10 月 19 日

**摘要**

本实验报告详述了在 ucore 操作系统内核中实现物理内存管理的全过程。实验的核心目标是为内核建立一个功能完备的物理内存管理子系统，并从物理地址模式过渡到分页虚拟地址模式。

首先，通过修改内核入口汇编代码 (`entry.S`)，我们构建了一个初始的顶级页表，利用 RISC-V Sv39 架构的大页特性，将内核映像高效地从物理地址空间映射到高位虚拟地址空间。随后，通过配置 `satp` 寄存器并刷新 TLB，成功启用了分页机制，为内核的后续运行奠定了虚拟内存环境的基础。

在此基础上，我们设计并实现了一个可扩展的物理内存管理器（PMM）框架。通过解析由 OpenSBI 传递的设备树（DTB），内核能够动态探测可用的物理内存范围。我们分析了 ucore 提供的默认 First-Fit 分配算法，并在此基础上，作为核心练习任务，自行设计并实现了 Best-Fit 连续物理内存分配算法。此外，我们还完成了对更高级的 Buddy System（伙伴系统）分配算法，slub 分配算法的实现与验证。报告详细阐述了这些算法的设计思想、数据结构、核心函数的实现逻辑，并通过设计的测试用例验证了其正确性，此外我们还仔细思考并分析了硬件的可用物理内存范围的获取方法。

本次实验成功地将操作系统原理中关于分页机制、多级页表、地址翻译以及多种动态内存分配策略等核心理论知识，与一个真实的内核开发实践相结合，极大地加深了对现代操作系统内存管理底层机制的理解。

# 目录

# 1 实验任务分工

- **梁景铭**: 负责扩展练习 Challenge：buddy system 分配算法的编程实现。
- **王一诺**: 负责扩展练习 Challenge：任意大小的内存单元 slub 分配算法的编程实现。
- **强博**: 负责练习 2：实现 Best-Fit 连续物理内存分配算法的编程实现。
- **共同完成**:
  - 对练习 2（Best-Fit 算法）的代码进行集体讨论、理解和改进。
  - 完成练习 1（理解 first-fit 算法）和扩展练习（硬件物理内存范围获取方法）的思考与分析。
  - 共同完成实验报告的撰写以及答辩的准备工作。

# 2 实验内容与目标

## 2.1 实验内容概述

本次实验在前一阶段构建的可启动系统的基础上，重点实现了操作系统的物理内存管理模块。主要内容包括：

1. **建立初始页表映射：**修改内核入口代码 (`entry.S`)，建立一个简单的页表，将内核代码和数据从物理地址空间映射到高位虚拟地址空间，并启用 Sv39 分页机制。
2. **物理内存探测与管理：**实现一个物理内存管理器 (`PMM`)，通过解析设备树 (DTB) 来探测可用的物理内存范围，并使用 `struct Page` 数据结构为每一个物理页帧建立元数据描述，为后续的内存分配做准备。
3. **分析 First-Fit 算法：**阅读并理解 ucore 中提供的默认物理内存分配算法——First-Fit 算法的实现原理，包括空闲链表的初始化、页的分配与释放、以及空闲块的合并等操作。
4. **实现 Best-Fit 算法：**基于对 PMM 框架和 First-Fit 算法的理解，参考 `default_pmm.c`，自行编程实现 Best-Fit 页面分配算法。

此外，实验还提供了 Buddy System、Slub 分配算法等 challenge 扩展练习。

## 2.2 核心学习目标

通过本次实验，旨在达成以下核心学习目标：

- **理解页表机制：**深入理解为何需要虚拟内存，掌握多级页表的原理、页表项 (PTE) 的结构，以及 CPU 如何通过 MMU 和页表完成虚拟地址到物理地址的翻译过程。
- **掌握物理内存管理方法：**学习操作系统如何发现、组织和管理物理内存。理解空闲链表法，并能通过数据结构 (`struct Page`, `free_area_t`) 来描述和追踪物理页帧的状态。

- **实践页面分配算法：**能够亲手实现经典的连续物理内存分配算法，如 First-Fit 和 Best-Fit，并理解它们的工作流程与潜在优缺点。
- **连接理论与实践：**将操作系统课程中关于存储管理的理论知识（如分段、分页、地址映射、内存分配策略等）与一个真实的、简化的内核实现对应起来，加深理解。

# 3 ucore 实验文档学习

在完成实验之前，我们首先对实验手册提供的背景知识进行了梳理和学习，核心知识点总结如下。

## 3.1 从直接物理寻址到虚拟内存抽象

### 3.1.1 直接物理地址访问的困境

在计算机系统的最底层，物理内存是由一系列具有唯一、确定地址的存储单元构成的硬件实体。CPU 通过地址总线发送这些物理地址来直接访问内存数据。在早期或简单的系统中，程序指令中包含的内存地址（例如 `MOV [0x80200000], %eax`）被直接解释为物理地址。这种**直接物理寻址**方式虽然直观，但在多道程序环境下会引发一系列难以克服的严重问题：

- **地址空间冲突与缺乏保护：**如果多个程序都试图访问相同的物理地址（例如 `0x80200000`），它们就会相互干扰，导致数据被意外修改、程序状态被破坏甚至系统崩溃。由于地址是直接暴露的，任何程序都可以无限制地访问任意物理内存，系统缺乏有效的内存隔离与保护机制。
- **内存分配的连续性要求：**每个程序都必须被加载到一块足够大的、**连续的**物理内存区域中才能运行。这导致了严重的**外部碎片**问题：随着程序的加载和释放，物理内存中会散布着许多不连续的小空闲块。即使这些小块的总容量足以容纳一个新程序，但由于找不到一个足够大的连续空间，新程序仍然无法加载。
- **程序重定位与共享困难：**程序的地址在链接时就被固定下来，这使得程序的加载和执行非常僵化。如果想在内存的不同位置运行同一个程序，或者让多个程序共享库代码，都将变得异常复杂。

这种对物理内存的直接、僵硬的管理方式，极大地限制了系统的效率、可靠性和灵活性。

### 3.1.2 虚拟内存

为了从根本上解决上述问题，现代操作系统引入了**虚拟内存**这一核心抽象。其本质是在程序与物理内存之间引入一个间接层，实现地址空间的解耦。程序不再直接与物理地址打交道，而是工作在一个操作系统为其构建的、私有的、看似连续的**虚拟地址空间**中。

这个过程可以精妙地类比为查一本为每个进程特制的“地址翻译词典”。程序生成的地址（虚拟地址）就像是词典中的一个词条，需要通过查询这本词典，才能找到其在物理内存中的真实位置（物理地址）。这本至关重要的“词典”，就是由操作系统管理、由硬件（内存管理单元 MMU）

使用的**页表**。通过为每个进程维护一套独立的页表，操作系统得以创造出多个隔离的、互不干扰的地址空间，即便底层只有一块共享的物理内存。

## 3.2 分页机制

### 3.2.1 为何不逐字节翻译？

如果对内存中的每一个字节都进行一对一的翻译，那么“地址翻译词典”本身将会无比庞大。例如，要映射 1MB 的内存空间，就需要一本至少存储 1MB 条目的词典，这在空间和时间开销上都是完全不可接受的。

### 3.2.2 页（Page）：以块为单位进行映射

程序的内存访问行为遵循著名的**局部性原理**，即在一段时间内，程序的访存行为会集中在某个局部区域。这启发我们可以将内存“打包”成块来管理和翻译，而不是逐字节进行。

据此，操作系统将虚拟地址空间和物理内存都划分为大小固定的、连续的块。虚拟地址空间中的块称为**页（Page）**，而物理内存中与之对应的、用于存放页的物理块称为**页帧**。地址翻译的基本单位是页，而非字节。同一页内的所有字节在虚拟地址和物理地址之间保持着固定的偏移关系。这种“批发式”的翻译方式，即**分页机制**，极大地降低了映射的复杂度，是现代虚拟内存系统的基石。

## 3.3 RISC-V Sv39 地址空间详解

本次实验所采用的 RISC-V 架构的 Sv39 分页模式，是当前主流的页式虚拟内存方案之一。

- **页面大小：**在 Sv39 模式下，标准的页面大小为 $2^{12} = 4096$ 字节，即 4KB。

### 3.3.1 地址结构与划分

在 Sv39 模式下，虚拟地址和物理地址有如下约定：

- **虚拟地址 (VA)**: 尽管在 64 位架构中，虚拟地址变量本身是 64 位的，但 Sv39 模式**仅使用其低 39 位**。硬件还有一个额外的规定：地址的第 63 位到第 39 位必须是第 38 位的**符号扩展**，否则该地址会被视为非法并引发异常。
- **物理地址 (PA)**: 物理地址最多可以使用 **56 位**。

一个 39 位的虚拟地址在逻辑上被划分为两部分：

- **虚拟页号 (VPN)**: 高 27 位 `[38:12]`。这 27 位是地址翻译的核心，用于在多级页表中进行索引，以最终找到对应的物理页帧号。
- **页内偏移 (Page Offset)**: 低 12 位 `[11:0]`。这 12 位可以唯一地索引一个 4KB 页面内的任意字节。在地址翻译过程中，页内偏移**保持不变**，直接复制到最终物理地址的对应低 12 位。

因此，地址翻译的本质任务就是：**将 27 位的虚拟页号（VPN）转换为一个 44 位物理页帧号（PPN）**。

### 3.3.2 分页机制的优势

通过页表建立虚拟页到物理页帧的映射关系，带来了巨大的灵活性和强大的功能：

- **内存使用的灵活性：**程序员和编译器面对的是一个巨大且连续的虚拟地址空间，无需关心底层物理内存是否是碎片化的。操作系统可以将一个程序的多个虚拟页映射到物理内存中任意离散的页帧上。
- **进程隔离与内存保护：**每个进程拥有独立的页表，确保了它们的虚拟地址空间相互隔离。一个进程无法直接访问另一个进程的物理内存，除非通过操作系统进行显式的共享设置。页表项中的权限位（读/写/执行）由硬件强制检查，提供了细粒度的内存保护。
- **高效的内存共享：**多个进程的页表中可以将不同的虚拟页映射到**同一个**物理页帧，从而轻松实现对代码或数据的共享，节约了物理内存。

## 3.4 内核启动与分页初始化

本次实验的核心任务是在 ucore 内核中实现物理内存管理。为此，我们对内核的启动流程进行了关键性的扩展：

1. **修改内核入口 `entry.S`:** 内核的第一站 `kern_entry` 函数，我们修改它来承担**设置初始虚拟内存环境**的重任。具体来说，它会构建一个最简单的页表，将内核自身映射到高位虚拟地址空间，然后将该页表的物理地址和 Sv39 模式位写入 `satp` 寄存器，从而**启用分页机制**。
2. **调用主控函数 `kern_init`:** 在分页机制成功开启后，控制权才被移交给 C 语言编写的内核主控函数 `kern_init`。
3. **初始化物理内存管理器:** `kern_init` 函数在完成一些基本的设备初始化后，其核心工作就是调用 `pmm_init` 函数，正式启动物理内存管理子系统。

## 3.5 物理内存管理（PMM）的核心步骤

`pmm_init` 函数组织了整个物理内存的发现、组织和管理过程，其步骤与操作系统原理紧密相连：

1. **探测物理内存资源：**首先，系统需要知道哪些物理地址范围是可用的 RAM。在 RISC-V 平台，这通常通过解析由启动加载器（如 OpenSBI）提供的设备树（DTB）来完成。
2. **建立页帧管理结构：**在确定可用内存后，操作系统以固定大小（4KB）的**页帧**为单位，将整个物理内存空间进行划分。并为每一个页帧创建一个元数据结构（在 ucore 中是 `struct Page`），用以追踪其状态：是空闲、已被分配，还是被系统保留。这对应了操作系统原理中的连续内存分配前的数据结构准备工作。
3. **建立页表并启动分页：**这是从物理地址模式向虚拟地址模式过渡的关键。操作系统利用前一步管理起来的空闲页帧，为内核构建一套完整的页表结构，精确描述虚拟页与物理页帧的对应关系。随后，通过将页表基地址载入 `satp` 寄存器，CPU 的 MMU 便开始根据这套映射规则进行地址翻译。

## 3.6 本次实验需解决的关键问题

综合上述流程，ucore 在实现物理内存管理时，需要解决两个核心的技术问题：

1. **如何建立虚拟地址与物理地址之间的联系？**这需要我们深刻理解多级页表的结构，并能够精确地构建页表项（PTE），以实现期望的地址空间布局（例如，将内核映射到高地址）。
2. **如何在现有 ucore 框架下实现物理内存页分配算法？**这要求我们基于 ucore 提供的 PMM 框架（`pmm_manager`），利用空闲链表等数据结构，实现具体的分配策略，如 First-Fit 或 Best-Fit。

接下来，我们将进一步分析完成 lab2 所需关注的关键技术细节和核心数据结构。

## 3.7 以页为单位管理物理内存

### 3.7.1 页表项

一个页表项（PTE）是描述单个虚拟页如何映射到物理页帧的基本数据单元。它就是“地址翻译词典”中的一个“词条”，存储在内存中，并由硬件 MMU 直接解析。为了保证查询效率，所有 PTE 都采用固定、统一的格式。

在 RISC-V Sv39 架构中，每个 PTE 占据 8 字节（64 位），其结构精心设计，包含了地址信息和一系列控制位：

| 63 54 | 53 10 | 9-8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | PPN[43:0] | RSW | D | A | G | U | X | W | R | V |

其中，第 53-10 位，共 44 位，构成了**物理页号**，它指向映射的目标物理页帧。而低 10 位则是一系列状态与权限标志位，精细地控制着对该页的访问行为：

- `V (Valid)`: **有效位**。这是最重要的标志位。若为 1，表示该 PTE 有效，MMU 可以正常使用它进行翻译；若为 0，表示这是一个无效的映射，任何通过此 PTE 的访问都会立即触发缺页异常，将控制权交给操作系统。
- `R, W, X`: **权限位**。分别控制对该页的读、写、执行权限。例如，若 `W` 位为 0，任何写操作都会触发保护性异常。这些位的组合定义了页面的访问属性：

| X | W | R | 含义 |
|---|---|---|---|
| 0 | 0 | 0 | 指向下一级页表的指针（非叶子节点） |
| 0 | 0 | 1 | 只读页面 |
| 0 | 1 | 1 | 可读可写页面 |
| 1 | 0 | 1 | 可读可执行页面 |
| 1 | 1 | 1 | 可读可写可执行页面 |
| 其他组合 | | | 保留，无效 |

- `U (User)`: **用户态访问位**。若为 1，表示用户态（U-Mode）代码可以访问该页；若为 0，则只有内核态（S-Mode）可以访问，从而实现了内核空间与用户空间的隔离。

- `G (Global)`: **全局映射位**。若为 1，表示该映射在所有地址空间中共享（例如内核代码），刷新 TLB 时可以不清空这类表项。
- `A (Accessed)`: **访问位**。一旦该页被访问（读、写或取指），硬件会自动将此位置 1。操作系统可以定期清零该位，通过检查它来判断哪些页面是活跃的，为页面置换算法提供依据。
- `D (Dirty)`: **脏位**。一旦该页被写入，硬件会自动将此位置 1。这对于页面换出至关重要：如果一个页面是“脏”的，意味着它在内存中的内容比在磁盘上的备份要新，换出前必须先写回磁盘。
- `RSW`: 两位，保留给操作系统自定义使用。

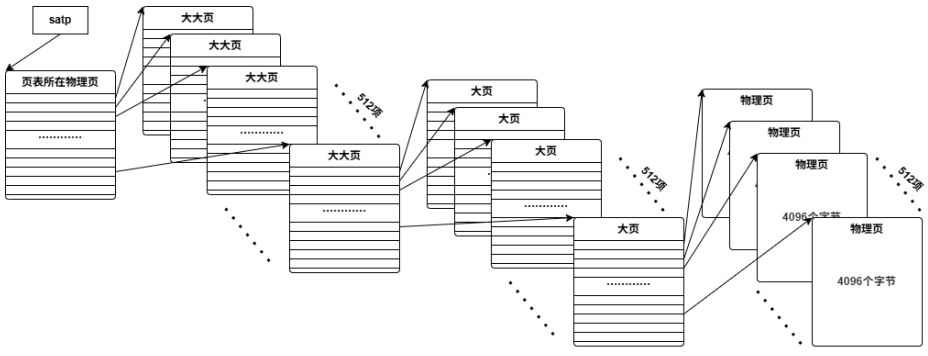
### 3.7.2 多级页表

为整个 39 位虚拟地址空间（$2^{27}$ 个虚拟页）建立一个扁平的、连续的页表是不可行的，因为它将占用整整 1GB 的内存（$2^{27} \times 8$ bytes）。然而，一个程序的虚拟地址空间通常是稀疏的，存在大片未使用的区域。

为了解决这个问题，Sv39 采用了**三级页表**结构，将线性的页表组织成一棵树。这种“分级”策略的核心思想是：只为实际使用的虚拟地址区域分配页表内存。如果一大片连续的虚拟地址都未被使用，那么在更高层级的页表中只需一个无效的 PTE 就可以代表整个区域，而无需为该区域内的每一个虚拟页都创建一个无效的 PTE。

在 Sv39 中，27 位的虚拟页号（VPN）被划分为三段 9 位的索引，分别对应三级页表的索引。地址翻译的过程就是一次从树根到叶子的遍历：

1. 使用 VPN 的高 9 位在**顶级（L2）页表**中找到一个 PTE。
2. 该 PTE 指向一个**二级（L1）页表**，再用 VPN 的中间 9 位在其中索引。
3. 得到的 PTE 又指向一个**末级（L0）页表**，最后用 VPN 的低 9 位在其中找到最终的叶子 PTE。




这个叶子 PTE 才包含了最终数据页的物理页号。由于每级页表的大小恰好是一个物理页（4KB），并且每个 PTE 为 8 字节，因此每级页表恰好可以容纳 $4096/8 = 512$ 个 PTE，正好对应 9 位索引（$2^9 = 512$）。此外，Sv39 还支持“大页”特性，即中间层级的 PTE 也可以直接指向一个大的数据块（2MB 或 1GB），从而减少 TLB 条目压力，提升性能。

### 3.7.3 页表基址寄存器（satp）

硬件 MMU 如何知道从哪里开始这趟“查表之旅”呢？RISC-V 架构为此提供了一个特殊的控制寄存器——`satp`。操作系统必须将顶级页表的**物理页号（PPN）**加载到这个寄存器中。

`satp` 寄存器的结构如下：

| 63 60 | 59 44 | 43 0 |
|---|---|---|
| MODE | ASID | PPN |

- `PPN`: 44 位的物理页号，指向 L2 页表的起始物理地址。
- `ASID` ：地址空间标识符，用于区分不同进程的地址空间，使得 TLB 可以同时缓存多个进程的映射。
- `MODE`: 控制地址翻译模式。设置为 `0000` 表示裸机模式（关闭分页），而设置为 `1000` 则表示启用我们所使用的 Sv39 分页模式。

通过修改 `satp` 寄存器的值，操作系统就可以在不同进程之间进行上下文切换，让 CPU “看向”不同的地址翻译“词典”，从而切换到不同的虚拟地址空间。

### 3.7.4 硬件加速：快表（TLB）

尽管多级页表巧妙地解决了页表自身的存储问题，但它引入了一个新的性能瓶颈：每次地址翻译都可能需要多次访问物理内存（在 Sv39 架构中最多三次），这被称为“页表遍历”。考虑到物理内存的访问速度远慢于 CPU 的时钟周期，这种额外的内存访问会极大地拖慢系统速度。

为了克服这一瓶颈，现代 CPU 架构引入了基于**局部性原理**的硬件优化。程序的内存访问行为通常表现出两种局部性：

- **时间局部性：**最近被访问过的内存地址，在不久的将来很可能再次被访问。
- **空间局部性：**一个内存地址被访问后，其附近的地址也很可能在不久后被访问。

这意味着，地址翻译的结果也具有高度的重复性。基于此，CPU 内部集成了一个专用的、高速的关联存储器，称为**快表 TLB)**。TLB 可以看作是页表项的一个硬件缓存，它存储了近期最常使用的虚拟页号（VPN）到物理页号（PPN）以及相关权限位的映射结果。

当进行地址翻译时，MMU 会首先并行地在 TLB 中查找当前虚拟地址对应的 VPN。

- **TLB 命中（Hit）：**如果在 TLB 中找到了匹配的条目，MMU 会立即从中获取物理地址和权限信息，从而跳过耗时的页表遍历过程。由于局部性原理，绝大多数地址翻译都能在 TLB 中命中，这使得虚拟内存的平均访问速度接近于直接物理寻址。
- **TLB 未命中（Miss）：**如果 TLB 中没有找到匹配项，硬件才会启动页表遍历，从内存中加载相应的 PTE。一旦加载成功，这个新的翻译结果就会被存入 TLB 中，以备后续使用。如果 TLB 已满，则会根据某种替换策略（如 LRU）替换掉一个旧的条目。

通过 TLB 这一硬件层面的缓存机制，操作系统虚拟内存的性能开销被降至最低，实现了高效的地址空间抽象。

## 3.8 UCore 分页机制实现详解

### 3.8.1 设计思路：建立段页式管理的关键考量

为了在 ucore 中成功实现分页机制，核心在于精确建立虚拟内存与物理内存之间的页映射关系，即正确构建三级页表。这个过程涉及复杂的硬件细节与多样的地址映射组合。总体而言，必须周全地考虑以下几个关键问题：

- 需要为哪些物理内存空间建立页映射关系？
- 具体的虚拟地址到物理地址的映射关系是什么？
- 各级页表的起始地址应设置在何处，需要多大的存储空间？
- 如何根据映射关系和权限要求，正确填充页目录项和页表项的内容？

### 3.8.2 内核空间映射的实现

在本项目中，我们的目标是将内核代码从物理地址空间（以 `0x80200000` 开始）映射到虚拟地址空间的高地址区域（以 `0xFFFFFFFFC0200000` 开始）。为了实现这一目标，首要步骤是修改链接器脚本 `tools/kernel.ld`，将内核的基地址设置为新的虚拟地址：

```
// tools/kernel.ld
BASE_ADDRESS = 0xFFFFFFFFC0200000; // 原为 0x80200000
```

仅仅修改链接脚本是不够的。在启动初期，CPU 仍处于 S 模式的裸机状态（Bare mode），它会将所有地址都当作物理地址来处理。此时，若直接跳转到链接器计算出的高位虚拟地址，将导致寻址错误。因此，我们必须在执行 C 代码之前，于汇编入口 `entry.S` 中抢先构建一个临时的页表，并启用分页机制，为内核的运行搭建好虚拟地址环境。

我们观察到，内核的虚拟地址与物理地址之间存在一个固定的偏移量：

$$\text{Offset} = \text{0xFFFFFFFFC0000000} - \text{0x80000000} = \text{0xFFFFFFFF40000000}$$

因此，对于内核空间内的任意虚拟地址 `va`，其对应的物理地址为 `pa = va - Offset`。我们的任务就是构造一个页表来实现这种线性映射。

考虑到内核大小不超过 1GiB，我们可以巧妙地利用 Sv39 的大页特性。只需分配一页 4KB 的内存作为顶级页表，并设置其最后一个 PTE（第 511 项），使其直接将 1GiB 的虚拟地址空间 `[0xFFFFFFFFC0000000, 0xFFFFFFFFFFFFFFFF]` 映射到物理地址空间 `[0x80000000, 0xC0000000)`。具体的汇编实现如下：

```
#include <mmu.h>
#include <memlayout.h>

    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
```

```
    # a0: hartid
    # a1: dtb physical address
    # save hartid and dtb address
    la t0, boot_hartid
    sd a0, 0(t0)
    la t0, boot_dtb
    sd a1, 0(t0)

    # t0 := 三级页表的虚拟地址
    lui     t0, %hi(boot_page_table_sv39)
    # t1 := 0xffffffff40000000 即虚实映射偏移量
    li      t1, 0xffffffffc0000000 - 0x80000000
    # t0 减去虚实映射偏移量 0xffffffff40000000, 变为三级页表的物理地址
    sub     t0, t0, t1
    # t0 >>= 12, 变为三级页表的物理页号
    srli    t0, t0, 12

    # t1 := 8 << 60, 设置 satp 的 MODE 字段为 Sv39
    li      t1, 8 << 60
    # 将刚才计算出的预设三级页表物理页号附加到 satp 中
    or      t0, t0, t1
    # 将算出的 t0(即新的 MODE| 页表基址物理页号) 覆盖到 satp 中
    csrw    satp, t0
    # 使用 sfence.vma 指令刷新 TLB
    sfence.vma
    # 从此，我们给内核搭建出了一个完美的虚拟内存空间!
    #nop # 可能映射的位置有些 bug。。插入一个 nop

    # 我们在虚拟内存空间中：随意将 sp 设置为虚拟地址!
    lui sp, %hi(bootstacktop)

    # 我们在虚拟内存空间中：随意跳转到虚拟地址!
    # 跳转到 kern_init
    lui t0, %hi(kern_init)
    addi t0, t0, %lo(kern_init)
    jr t0

.section .data
    # .align 2^12
    .align PGSHIFT
```

```
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:

.section .data
    # 由于我们要把这个页表放到一个页里面，因此必须 12 位对齐
    .align PGSHIFT
    .global boot_page_table_sv39
# 分配 4KiB 内存给预设的三级页表
boot_page_table_sv39:
    # 0xffffffff_c0000000 map to 0x80000000 (1G)
    # 前 511 个页表项均设置为 0 ，因此 V=0 ，意味着是空的 (unmapped)
    .zero 8 * 511
    # 设置最后一个页表项，PPN=0x80000，标志位 VRWXAD 均为 1
    .quad (0x80000 << 10) | 0xcf # VRWXAD

    .global boot_hartid
boot_hartid:
    .quad 0
    .global boot_dtb
boot_dtb:
    .quad 0
```

总结而言，进入虚拟内存访问方式的核心三步骤是：

1. 分配并初始化页表内存空间。
2. 设置页表基址寄存器（satp）。
3. 刷新 TLB（通过 sfence.vma 指令）。

完成这三步后，CPU 便成功进入了分页模式，为后续复杂的内存管理打下了坚实的基础。

# 4 物理内存管理实现

## 4.1 设计思路与核心抽象

在虚拟内存的宏伟蓝图之下，是对物理内存（PMM）精细而高效的管理。PMM 是整个内存管理体系的基石，它必须为上层（如虚拟内存管理器）提供一套清晰可靠的服务接口，主要包括：

- **分配物理页：** 根据请求分配一个或多个连续的物理页帧。
- **释放物理页：** 回收不再使用的页帧，并将其归还到空闲池中。

- **查询空闲页数：**返回当前系统中可用的物理页帧总数。

为了实现这种设计的模块化和可扩展性，ucore 采用了一种优雅的抽象，即 `struct pmm_manager`。这个结构体通过函数指针，定义了一个通用物理内存管理器的行为规范。任何一种具体的分配算法（如 First-Fit, Best-Fit, Buddy System），只需按照这个接口实现相应的函数，就可以无缝地“插入”到 ucore 内核中。

```
struct pmm_manager {
    const char *name;  // 管理器名称
    void (*init)(void); // 初始化管理器内部数据结构
    void (*init_memmap)(struct Page *base, size_t n); // 将一块物理内存加入管理
    struct Page *(*alloc_pages)(size_t n); // 分配 n 个页
    void (*free_pages)(struct Page *base, size_t n); // 释放 n 个页
    size_t (*nr_free_pages)(void); // 返回空闲页数
    void (*check)(void); // 自检函数
};
```

为了追踪每一个物理页帧的状态，ucore 定义了核心数据结构 `struct Page`。系统中的每一个物理页帧都唯一对应一个 `struct Page` 实例。该结构体不仅通过 `ref` 记录了页面的引用计数，还通过 `flags` 标志位（如 `PG_reserved`, `PG_property`）来描述页面的保留状态或其是否为空闲块的首页。特别地，`page_link` 成员是一个 `list_entry_t` 类型的双向链表节点，它使得空闲的 `Page` 结构可以被组织成高效的空闲链表。

## 4.2 内存发现与初始化流程

ucore 的物理内存管理始于内核主函数 `kern_init` 对 `pmm_init` 的调用。

```
// kern/init/init.c
int kern_init(void) {
    // ... 其他初始化 ...
    pmm_init();  // 初始化物理内存管理
    // ...
}
```

`pmm_init` 函数协调 PMM 子系统的整个设置过程。

```
// kern/mm/pmm.c
void pmm_init(void) {
    init_pmm_manager(); // 1. 选择并初始化一个具体的 PMM 算法
    page_init();        // 2. 探测物理内存并建立页帧元数据
    check_alloc_page(); // 3. 运行自检程序验证 PMM 的正确性
    // ...
}
```

这个过程的核心在于 page_init 函数。它的首要任务是**发现可用的物理内存**。在 RISC-V 平台，这一信息由引导加载器（OpenSBI）通过设备树传递给内核。kern_entry 会保存 DTB 的物理地址，随后 dtb_init 函数负责解析它，提取出物理内存的基地址和大小。

拿到内存范围后，page_init 开始了精密的布局工作。它首先确定了整个物理内存空间需要多少个 struct Page 来管理（一个页帧对应一个），然后在内核代码和数据段之后，紧接着找到一块对齐的内存区域，用于存放这个庞大的 pages 数组。接着，它将已被 OpenSBI、内核自身以及 pages 数组所占用的所有物理页帧，通过设置 PG_reserved 标志位进行“保留”，确保它们不会被后续的分配器错误地分配出去。最后，将剩余的、真正可用于动态分配的连续物理内存区域，通过调用当前 pmm_manager 的 init_memmap 函数，正式移交给具体的分配算法来管理。

### 4.3 First-Fit 算法实现解析

ucore 的默认物理内存管理器是 default_pmm_manager，它实现了经典的 **First-Fit** 算法。该算法通过一个按地址升序排列的双向链表 free_list 来管理所有空闲内存块。

- **初始化 (default_init / default_init_memmap):** default_init 简单地初始化一个空的 free_list。当 page_init 发现一块大的可用内存时，会调用 default_init_memmap，该函数将这块内存作为一个单独的、巨大的空闲块，插入到 free_list 中。
- **分配 (default_alloc_pages):** 当请求分配 n 个页面时，算法从 free_list 的头部开始线性扫描。它会寻找**第一个**大小大于或等于 n 的空闲块。找到后，如果该块的大小恰好等于 n，就将其从链表中移除；如果大于 n，则将其**分裂**：前 n 页作为分配结果返回，剩余部分形成一个新的、更小的空闲块，并留在链表中。
- **释放 (default_free_pages):** 当一块内存被释放时，它被视为一个新的空闲块，并按地址顺序插入到 free_list 中。插入后，算法会立即检查其在链表中的**前一个和后一个**邻居。如果发现任何一个邻居在物理地址上与之是连续的，就会执行**合并**操作，将它们融合成一个更大的空闲块。这个机制是至关重要的，因为它能有效地对抗内存碎片化。

```
// kern/mm/default_pmm.c

// ... (init 和 init_memmap 函数) ...

static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
```

```
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) { // 找到了第一个足够大的块
            page = p;
            break;
        }
    }
    if (page != NULL) {
        if (page->property > n) { // 如果块大于所需，则分裂
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add_before(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link)); // 从空闲链表中移除
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}

static void
default_free_pages(struct Page *base, size_t n) {
    // ... (初始化新释放的块) ...

    // 按地址插入链表
    // ...

    // 向前合并
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    // 向后合并
```

```
        le = list_next(&(base->page_link));
        if (le != &free_list) {
            p = le2page(le, page_link);
            if (base + base->property == p) {
                base->property += p->property;
                ClearPageProperty(p);
                list_del(&(p->page_link));
            }
        }
    }
```

总而言之，ucore 的物理内存管理通过一个灵活的管理器接口，成功地将内存发现、元数据建立和具体的分配策略解耦。其默认的 First-Fit 实现虽然简单，但完整地展示了动态内存分配中的查找、分裂和合并等核心操作，为整个操作系统提供了坚实的内存基础。

# 5 实验练习解答

## 5.1 练习 1：理解 first-fit 连续物理内存分配算法

### 5.1.1 问题重述

请仔细阅读实验手册的教程并结合 `kern/mm/default_pmm.c` 中的相关代码，认真分析 `default_init`，`default_init_memmap`，`default_alloc_pages`，`default_free_pages` 等相关函数，并描述程序在进行物理内存分配的过程以及各个函数的作用。并回答问题：你的 first fit 算法是否有进一步的改进空间?

### 5.1.2 设计实现过程

练习 1 的代码设计实现了一个 first-fit 思想的内存分配算法，即在分配内存时找到第一个满足内存分配条件的物理页进行分配。具体实现代码解析如下：

首先是初始化函数 `default_init`，对空闲链表进行了初始化，并将空闲链表中的空闲页数属性先置为 0。

函数 `default_init_memmap` 实现了初始化内存映射的过程，首先将每个物理页面初始化，保证页面处于空闲状态，然后按照地址顺序将空闲物理页面插入到空闲链表当中，同步维护空闲页数 `nr_free`

函数 `default_alloc_pages` 实现了物理页面分配的功能，首先检查是否有足够的空闲页进行分配，如果有的化就从空闲链表的第一个节点开始遍历，找到符合要求的空闲页并对其进行分割，分割后剩余大小的空间重新插回空闲链表中并更新空闲页数 `nr_free`

函数 `default_free_pages` 实现了释放页面的功能，首先清空需要被释放的页面，然后将释放后的空闲块进行合并，检查前后的空闲块是否由连续的可以被合并如果有，就进行合并，如果没有，就直接按照地址顺序插入到链表当中。

### 5.1.3 问题解答

当前实现的 first fit 分配算法在具体操作过程中会在内存中产生很多无法被分配的小块的内存碎片，使得内存的使用效率低下，并且每次都从低地址空间开始检索，分配效率低下。

可以根据块的大小维护多个空闲链表，在分配内存的时候根据需要从不同的空闲链表进行检索并更新，提高了内存分配的效率。

## 5.2 练习 2：实现 Best-Fit 连续物理内存分配算法

### 5.2.1 问题重述

参考 `kern/mm/default_pmm.c` 对 First Fit 算法的实现，编程实现 Best Fit 页面分配算法。请在实验报告中简要说明你的设计实现过程，阐述代码是如何对物理内存进行分配和释放，并回答问题：你的 Best-Fit 算法是否有进一步的改进空间?

### 5.2.2 设计实现过程

best-fit 算法的核心思想是在分配内存时，从所有满足需求的空闲块中选择大小最接近的内存块进行分配，通过这种思路可以有效的减少内存碎片的产生和大小。在设计与实现过程中，我主要完成了以下几个关键函数：

**1. 内存初始化 - `best_fit_init_memmap`** 该函数用于初始化一个空闲内存块，主要完成以下工作：

- 遍历要初始化的所有页框，清空标志位和引用计数
- 设置基页的属性：`base->property = n` 记录块大小，`SetPageProperty(base)` 标记为空闲状态
- 按物理地址顺序将空闲块插入链表，为后续合并操作创造条件

关键实现代码：

```
// 按地址顺序插入链表
if (base < page) {
    list_add_before(le, &(base->page_link));
    break;
} else if (list_next(le) == &free_list) {
    list_add(le, &(base->page_link));
}
```

**2. 页面分配 - `best_fit_alloc_pages`** 这是 Best-Fit 算法的核心函数，实现流程如下：

- 遍历整个空闲链表，找到所有大小 ≥ 请求大小 n 的空闲块
- 记录其中大小最小的合适块（即最接近 n 的块）

- 如果找到合适块，从链表中删除该块
- 如果块大小大于需求，分割剩余部分重新插入链表
- 更新空闲页计数，清除分配页的属性标记

关键实现代码：

```
// 寻找最小合适的空闲块
while ((le = list_next(le)) != &free_list) {
    struct Page *p = le2page(le, page_link);
    if (p->property >= n && p->property < min_size) {
        page = p;
        min_size = p->property; // 更新最小大小
    }
}
```

**3. 页面释放 - `best_fit_free_pages`** 释放页面的主要工作包括：

- 初始化释放页的属性，设置块大小和空闲标记
- 按地址顺序将释放块插入空闲链表
- 执行前后合并检查：如果与相邻空闲块连续，则合并成大块
- 更新空闲页计数

关键实现代码：

```
// 前向合并
if (p + p->property == base) {
    p->property += base->property;
    ClearPageProperty(base);
    list_del(&(base->page_link));
    base = p; // 更新基指针便于后向合并检查
}

// 后向合并
if (base + base->property == p) {
    base->property += p->property;
    ClearPageProperty(p);
    list_del(&(p->page_link));
}
```

### 5.2.3 问题解答

我们认为目前实现的 best-fit 算法在以下方面还有改进空间：

**1. 时间复杂度优化　当前问题**：每次分配都需要遍历整个链表，时间复杂度为 O(N) **改进方案**：

- **平衡二叉树**：使用搜索树数据结构维护空闲块，搜索复杂度降为 O(logN)

**2. 缓存和预分配　改进方案**：

- **常用大小缓存**：维护最近释放的常用大小块缓存
- **预分配池**：预分配一批常用大小的页块，减少实时分配开销
- **分配预测**：基于历史分配模式预测未来需求

## 5.3 扩展练习 Challenge：buddy system（伙伴系统）分配算法

### 5.3.1 问题重述

实现 Buddy System 算法，将可用存储空间划分为大小为 2 的 n 次幂的存储块进行管理。

### 5.3.2 核心思想与数据结构

伙伴系统（Buddy System）是一种经典的内存分配算法，其本质是**“分离适配”**策略的一种高度优化的特例。它通过一种严谨而高效的方式来管理内存，以期在分配和回收速度与内存碎片之间取得良好的平衡。其核心思想在于，将整个待管理的内存空间（大小必须是 2 的幂）视为一个整体，当需要分配时，若无大小恰好的空闲块，则递归地将一个更大的块**对半分裂**成两个大小相等的“伙伴”，直至产生满足需求的最小块。反之，当一个块被释放时，算法会检查其伙伴是否也空闲，若是，则立即将二者**合并**成一个更大的父块，并尝试继续向上合并。

为了高效地追踪这种严谨的伙伴关系和分裂合并操作，**完全二叉树**成为了最理想的底层数据结构。这棵树的节点与内存块一一对应：根节点代表整个内存区域，每个内部节点代表一个内存块，其子节点则是该块分裂后的两个伙伴，而叶子节点则代表了系统能管理的最小粒度的内存块。通过将这棵树以数组形式存储，可以利用下标运算快速定位任何节点的父节点、子节点乃至伙伴节点，极大地提升了管理效率。

### 5.3.3 内存分配流程

当应用程序请求分配大小为 `size` 的内存时，伙伴系统的分配流程体现了其“分裂”的核心机制。首先，算法会将请求大小 `size` **向上取整到最接近的 2 的幂**，得到实际需要分配的块大小 `block_size`。例如，请求 70K 内存将被适配为 128K，这也是伙伴系统会产生**内部碎片**的根源。接着，算法会直接在管理 `block_size` 大小块的空闲链表（或树的对应层级）中查找可用块。如果找到，便直接分配，流程结束。

若当前层级没有空闲块，算法的精髓便得以展现：它会**向上一层级**（即大小为 `2 * block_size` 的块）寻找。一旦找到一个更大的空闲块，就将其从其所属的空闲列表中取出，并**分裂成两个 `block_size` 大小的伙伴**。其中一个伙伴被分配给应用程序，而另一个则被加入到 `block_size` 对应的空闲链表中备用。如果上一层级仍然没有，就继续向上递归查找，直至找到可用块并逐级分裂下来。在分裂过程中，每一步产生的“另一半”伙伴都会被添加到相应大小的空闲池中。如果直到根节点都无法满足需求，则宣告分配失败。

### 5.3.4 内存释放与合并流程

当应用程序释放一个内存块时，其流程是分配的逆过程，核心在于“合并”。释放操作首先将被释放的块标记为空闲，并归还到对应大小的空闲池中。随后，算法的关键步骤——**检查伙伴状态**——便开始了。通过被释放块的地址和大小，可以精确计算出其伙伴块的地址，并检查其是否也处于空闲状态。

如果伙伴块恰好也为空闲，算法会立即将这对伙伴**合并成一个两倍大的父块**，并将原先的两个小块从它们的空闲池中移除。这个新形成的、更大的父块，会接着**递归地触发新一轮的合并检查**，即它会继续检查自己的“新伙伴”是否空闲。这个递归合并的过程会一直持续，直到遇到一个非空闲的伙伴，或者已经合并到根节点（代表整个内存区域都已回收）。这种积极的、递归的合并策略是伙伴系统的精髓所在，它确保了只要有可能，系统就会自动恢复出最大的连续空闲内存块，从而为未来的大内存请求做好准备。

### 5.3.5 设计实现与代码解析

本次实现严格遵循伙伴系统的核心原理，通过一系列数据结构和辅助函数来管理物理内存。

#### 数据结构

- `BUDDY_MAX_ORDER`: 定义了伙伴系统管理的最大阶数。在本实现中为 11，意味着最小块为 $2^0 = 1$ 页，最大块为 $2^{10} = 1024$ 页。
- `free_area[BUDDY_MAX_ORDER]`: 这是一个核心的数组，数组的每一个元素都是一个链表头。`free_area[i]` 代表了所有大小为 $2^i$ 页的空闲块组成的链表。
- `buddy_base` 和 `buddy_npages`: 记录了此伙伴系统实例所管理的全部物理页帧的起始 `Page` 结构指针和总页数。

#### 核心函数解析

- `buddy_init_memmap`: 这是初始化的关键。它接收一块连续的物理内存，并将其“雕刻”成符合伙伴系统规范的初始空闲块。它通过一个循环，从起始地址开始，贪心地切出最大的、地址对齐的 $2^k$ 大小的块，并将其加入到对应阶的 `free_area` 链表中。
- `buddy_alloc_pages`:
  1. 首先，通过 `ceil_order` 将请求的页数 `n` 向上适配到最接近的 2 的幂阶 `need`。
  2. 从 `need` 阶开始，向上查找第一个非空的 `free_area` 链表。
  3. 如果找到的阶 `o` 大于 `need`，则进入分裂流程：将从 `o` 阶取出的块逐级对半分裂，直到得到一个 `need` 阶的块。在每一级分裂中，分裂出的“右半边”伙伴块会被加入到对应阶的空闲链表中。
  4. 最终，将大小适配的块从链表中移除，更新空闲页计数，并返回其指针。
- `buddy_free_pages`:

1. 将要释放的内存（可能包含多个不同大小的对齐块）进行分解，逐块处理。
2. 对于每一个要释放的块，进入一个循环，进行递归合并的尝试。
3. 在循环中，使用 `buddy_idx_of` 计算当前块的伙伴块的索引。
4. 通过 `find_free_block` 在对应阶的空闲链表中查找这个伙伴。
5. 如果伙伴块也为空闲，则将伙伴块从其链表中移除，并将当前块的大小翻倍（阶数加一），然后继续循环，尝试与新的、更大尺寸的伙伴合并。
6. 如果伙伴不空闲，则合并过程终止。将最终（可能已经合并了多次）的块加入到其对应阶的空闲链表中。

#### 5.3.6 测试用例与正确性验证

为验证算法的正确性，我们设计了一个名为 `buddy_check` 的测试函数，该函数模拟了一系列精心设计的内存分配和释放请求，并检查最终状态是否符合预期。测试序列与参考图示中的场景完全对应。

测试场景基于一个总大小为 1024 页的内存池，具体操作序列如下：

1. **分配 A, B, C, D:**
   - A 请求 32 页，直接分配 order=5 的块 [0, 32)。
   - B 请求 64 页，分配 order=6 的块 [64, 128)。
   - C 请求 60 页 (向上取整为 64)，分配 order=6 的块 [128, 192)。
   - D 请求 150 页 (向上取整为 256)，分配 order=8 的块 [256, 512)。

   此时，初始的 1024 页块已被分裂，内存布局如图**??**上半部分所示。

2. **释放 B:** 释放块 [64, 128)。此时其伙伴 [0, 64) 并非完全空闲（被 A 占用），因此无法立即合并。

3. **释放 A:** 释放块 [0, 32)。此时，它首先会检查其 order=5 的伙伴 [32, 64) 是否空闲。由于 [32, 64) 从未被分配，是空闲的，二者合并成 order=6 的块 [0, 64)。接着，这个新形成的块 [0, 64) 会继续检查其 order=6 的伙伴 [64, 128)（即刚刚释放的 B），发现也为空闲，于是再次合并，最终形成一个 order=7 的大空闲块 [0, 128)。这个过程完美地验证了递归合并的逻辑。

4. **分配 E, F:**
   - E 请求 100 页 (向上取整为 128)，此时刚好有合并而成的 [0, 128) 空闲块，直接分配。
   - F 请求 100 页 (向上取整为 128)，此时需要从 [512, 1024) 这个 order=9 的块中分裂，最终得到 [512, 640)。

测试函数的最后，通过打印所有分配的块的最终位置和大小，并计算内部碎片，可以清晰地看到每一步操作的结果都与伙伴算法的理论行为完全一致，从而证明了我们实现的正确性。

## 5.4 扩展练习 Challenge：任意大小的内存单元 slub 分配算法

### 5.4.1 问题重述

实现一个简化的 slub 算法，实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现，能够体现其主体思想即可。

### 5.4.2 总体框架与设计目标

- 将物理内存管理拆分为“页层 + slab 对象层”两级，既保留页粒度的灵活性，又引入 SLUB 的高效对象缓存。
- 页层沿用 best_fit 策略，负责提供/回收连续页；对象层以单页 slab 为单位缓存特定尺寸对象，提高小对象分配效率。
- 分配时根据对象尺寸选择默认或自定义 cache → 如需新 slab 就向页层要 1 页 → 从 free_-head 取槽位；释放时写回索引，必要时把 slab 从 full 挂回 partial，完全空闲则整页回收。这种 slab 化策略让同尺寸运行时复用同一页，避免反复在页层拆分。
- 通过 pmm_manager 接口对外提供统一入口，并内置 check。

### 5.4.3 设计实现过程

**数据结构定义** 定义 slab 与 cache 元数据:

- 用 struct slub_slab 驻留在每一个单页 slab 开头，记录所属 cache、空闲链、容量等信息。
- 用 struct slub_cache 描述不同对象尺寸的缓存实例，并维护 partial/full slab 列表。

```
1  struct slub_slab {
2        uint32_t magic;              // 用于基本合法性校验
3        struct slub_cache *cache;    // 指向所属的对象缓存
4        struct Page *page;           // 物理页描述符，便于回收
5        uint16_t free_count;         // 当前剩余空闲对象数
6        uint16_t capacity;           // slab 可容纳的对象总数
7        uint16_t free_head;          // 空闲对象链表头（保存的是索引）
8        uint16_t obj_stride;         // 对象步长（包含对齐后大小）
9        list_entry_t link;           // 串入 cache 的 partial/full 链表
10   };
11
12   struct slub_cache {
13       const char *name;            // 缓存名称
14       size_t obj_size;             // 用户请求的对象大小
15       size_t obj_stride;           // 实际分配的步长（包含对齐和 freelist
             空间）
16       size_t align;                // 对齐要求
17       uint16_t objs_per_slab;      // 每个 slab 可容纳的对象数量
18       size_t slabs_total;          // 当前总共持有的 slab 数
```

```
    size_t slabs_partial;        // 位于 partial 链表中的 slab 数
    size_t inuse_objs;           // 已分配但未释放的对象总数
    bool is_default;             // 是否为默认 size class
    bool active;                 // 缓存是否处于激活状态
    list_entry_t node;           // 串入全局 cache 链表
    list_entry_t partial;        // 局部链表：仍有空闲对象的 slab
    list_entry_t full;           // 局部链表：满载的 slab
};
```

**页层**　页层继续沿用 best_fit 的设计策略，此处不再赘述。

**cache 初始化与管理**　:cache 结构体设计

- **cache 元数据搭建**: 初始化 cache 结构体，计算对齐、步长、单页容纳对象数，挂接到全局链表。

```
static void slub_cache_setup(struct slub_cache *cache, const char *name,
                             size_t obj_size, size_t align, bool is_default) {
    memset(cache, 0, sizeof(*cache));
    cache->name = name;
    cache->obj_size = obj_size;
    cache->align = (align == 0 ? SLUB_MIN_ALIGN : align);
    if (cache->align < sizeof(uint16_t)) {
        cache->align = sizeof(uint16_t);
    }
    size_t stride = obj_size;
    if (stride < sizeof(uint16_t)) {
        stride = sizeof(uint16_t);
    }
    stride = ROUNDUP(stride, cache->align);
    cache->obj_stride = stride;
    size_t usable = PGSIZE - sizeof(struct slub_slab);
    cache->objs_per_slab = (usable >= stride)
                           ? (uint16_t)(usable / stride)
                           : 0;
    cache->slabs_total = 0;
    cache->slabs_partial = 0;
    cache->inuse_objs = 0;
    cache->is_default = is_default;
    cache->active = 1;
    list_init(&cache->node);
    list_init(&cache->partial);
    list_init(&cache->full);
```

```
28        list_add(&slub_cache_list, &cache->node);
29    }
```

默认对齐至少 8 字节，同时保证 obj_stride 至少能容纳 uint16_t freelist 指针。

可用容量 = PGSIZE - sizeof(struct slub_slab)，这决定每个 slab 能放多少个对象。

将 cache 加入 slub_cache_list，方便统计和调试。

slabs_partial/slabs_total 等统计字段清零，为后续维护做铺垫。

- **默认 size class**：在初始化阶段注册常见的 12 个对象尺寸，避免运行期动态创建导致的开销。

```
1  static void slub_bootstrap_default_caches(void) {
2        for (size_t i = 0; i < ARRAY_SIZE(slub_default_caches); i++) {
3            struct slub_cache *cache = &slub_default_caches[i];
4            snprintf(slub_default_names[i],
5                     sizeof(slub_default_names[i]),
6                     "slub-%u", (unsigned)slub_default_sizes[i]);
7            slub_cache_setup(cache, slub_default_names[i],
8                             slub_default_sizes[i], SLUB_MIN_ALIGN, 1);
9        }
10   }
```

默认尺寸覆盖 8 到 768 字节常用范围，满足实验中内核对象的绝大部分需求。

同时进行了命名如 slub-128，便于日志和调试。

需要注意，此处仅初始化 cache 元数据，不会提前申请 slab，避免系统刚启动就消耗多页。

**slub_slab 生命周期管理**

- **创建 slab** 按需向页层申请 1 页，初始化 slub_slab 元数据和空闲链，将 slab 挂到 cache 的 partial 链表。

```
1  static struct slub_slab *slub_new_slab(struct slub_cache *cache) {
2        SLUB_ASSERT(cache->objs_per_slab > 0);
3        struct Page *page = slub_page_alloc(1);
4        if (page == NULL) {
5            return NULL;
6        }
7        struct slub_slab *slab = (struct slub_slab *)page2kva_local(page)
             ;
8        memset(slab, 0, sizeof(*slab));
9        slab->magic = SLUB_SLAB_MAGIC;
10       slab->cache = cache;
11       slab->page = page;
12       slab->capacity = cache->objs_per_slab;
```

```
13        slab->free_count = slab->capacity;
14        slab->free_head = (slab->capacity == 0) ? SLUB_FREELIST_END : 0;
15        slab->obj_stride = (uint16_t)cache->obj_stride;
16        list_init(&slab->link);
17        uint8_t *base = slab_obj_base(slab);
18        for (uint16_t i = 0; i < slab->capacity; i++) {
19            uint8_t *slot = base + i * slab->obj_stride;
20            uint16_t next = (i + 1 < slab->capacity) ? (i + 1) :
                  SLUB_FREELIST_END;
21            *((uint16_t *)slot) = next;
22        }
23        cache->slabs_total++;
24        cache->slabs_partial++;
25        list_add(&cache->partial, &slab->link);
26        return slab;
27    }
```

通过 page2kva_local 得到页的内核地址，直接把 struct slub_slab 放在页首。

free_head 初始指向 0，说明第一个对象槽为 0 号索引。

初始化循环将每个槽的前 2 字节写成“下一个空闲槽的索引”，天然形成单向链表。

将 slab 纳入 partial 链表，并同步更新 slabs_total/slabs_partial。

- **释放 slab**：当 slab 内所有对象均被释放时，移除 slab 并归还页层，使得内存即时回收。

```
1  static void slub_release_slab(struct slub_slab *slab) {
2        SLUB_ASSERT(slab->magic == SLUB_SLAB_MAGIC);
3        struct slub_cache *cache = slab->cache;
4        slab->magic = 0;
5        SLUB_ASSERT(slab->free_count == slab->capacity);
6        SLUB_ASSERT(cache->slabs_total > 0);
7        cache->slabs_total--;
8        slub_page_free(slab->page, 1);
9    }
```

通过 magic 和 free_count == capacity 双重检查确保 slab 处于全空闲状态。

归还页层后，partial 链中对该 slab 的引用已在 slub_cache_do_free 中解除。

**对象分配与释放**

- **分配对象**在 slub_alloc() 中匹配合适 cache，必要时创建新 slab，再从空闲链中取出对象。超出范围则返回 NULL。

```
1  void *slub_alloc(size_t size) {
2        if (size == 0) {
3            return NULL;
```

```
    }
    struct slub_cache *cache = slub_select_cache(size);
    if (cache != NULL) {
        return slub_cache_do_alloc(cache);
    }
    /* 设计约束：当前实现仅覆盖单页 SLUB 缓存，不支持跨页大对象。 */
    return NULL;
}

static void *slub_cache_do_alloc(struct slub_cache *cache) {
    if (cache->objs_per_slab == 0) {
        return NULL;
    }
    if (list_empty(&cache->partial)) {
        if (slub_new_slab(cache) == NULL) {
            return NULL;
        }
    }
    list_entry_t *le = list_next(&cache->partial);
    struct slub_slab *slab = le2slab(le);
    SLUB_ASSERT(slab->free_count > 0);
    uint16_t obj_index = slab->free_head;
    uint8_t *slot = slab_obj_base(slab) + obj_index * slab->
       obj_stride;
    slab->free_head = *((uint16_t *)slot);
    slab->free_count--;
    cache->inuse_objs++;
    if (slab->free_count == 0) {
        list_del(&slab->link);
        list_add(&cache->full, &slab->link);
        SLUB_ASSERT(cache->slabs_partial > 0);
        cache->slabs_partial--;
    }
    memset(slot, 0, cache->obj_size);
    return slot;
}
```

slub_select_cache 会优先使用默认 cache，如果尺寸未覆盖则直接返回 NULL，方便上层做降级处理。

slub_cache_do_alloc 保证了对 partial 链的原子操作：新 slab 创建后立即挂入 partial，队首被取空则转移到 full。

返回槽位前用 memset 清零用户可见区域，屏蔽 freelist 指针残留。

- **释放对象**：slub_free 校验 magic → 找到 slab → 写回 freelist → 维护 partial/full 链 → slab 全

空闲时归还页层。

```
void slub_free(void *ptr) {
      if (ptr == NULL) {
          return;
      }
      struct Page *page = kva2page_local(ptr);
      void *base = page2kva_local(page);
      uint32_t magic = *((uint32_t *)base);
      SLUB_ASSERT(magic == SLUB_SLAB_MAGIC);
      struct slub_slab *slab = (struct slub_slab *)base;
      slub_cache_do_free(slab->cache, slab, ptr);
  }

  static void slub_cache_do_free(struct slub_cache *cache, struct
     slub_slab *slab,
                                 void *obj) {
      uint8_t *base = slab_obj_base(slab);
      uintptr_t offset = (uint8_t *)obj - base;
      SLUB_ASSERT(offset % slab->obj_stride == 0);
      uint16_t idx = offset / slab->obj_stride;
      *((uint16_t *)obj) = slab->free_head;
      slab->free_head = idx;
      slab->free_count++;
      SLUB_ASSERT(cache->inuse_objs > 0);
      cache->inuse_objs--;
      if (slab->free_count == 1) {
          list_del(&slab->link);
          list_add(&cache->partial, &slab->link);
          cache->slabs_partial++;
      }
      if (slab->free_count == slab->capacity) {
          list_del(&slab->link);
          SLUB_ASSERT(cache->slabs_partial > 0);
          cache->slabs_partial--;
          slub_release_slab(slab);
      }
  }
```

通过页首 magic 可以快速判断是否为合法的 slab 对象，防止跨 cache 释放或野指针误用。

释放的对象前 2 字节写入原 free_head，新的空闲链头就是该对象索引。

一旦 slab 重新变为部分空闲，需要把它从 full 移到 partial；当完全空闲时，立即归还页层，避免占用整页。

**关于 check** 设计了五类测试验证页层、对象层、通用接口与压力场景的行为，确保回归安全。

```
static void slub_page_basic_check(void) { ... }
  static void slub_page_fragment_check(void) { ... }
  static void slub_cache_basic_check(void) { ... }
  static void slub_alloc_general_check(void) { ... }
  static void slub_stress_check(void) { ... }

  static void slub_check(void) {
      slub_page_basic_check();
      slub_page_fragment_check();
      slub_cache_basic_check();
      slub_alloc_general_check();
      slub_stress_check();
      SLUB_LOG("all␣checks␣passed\n");
  }
```

- 基础页测试：连续申请三个单页并释放，确认最基本的页层操作正确
- 碎片合并：将一个 8 页块拆成两段释放（前 3+ 后 5），此举制造“碎片”；随后再申请 8 页，期望能重新拿到原始块（说明成功把碎片合并回来）。
- 缓存测试：创建一个 64 字节的自定义 cache，批量分配 objs_per_slab * 2 个对象；释放一半，再释放剩余所有，最后要求 cache->slabs_total == 0（所有 slab 被回收）。
- 对象申请释放测试：准备多种大小（1 511 字节），循环调用 slub_alloc 和 slub_free，并在前后读 slub_page_nr_free()，要求释放后空闲页总数与之前相等。
- 压力测试：批量申请/释放 48 字节对象，检验高频场景稳定性。

#### 5.4.4 与其他算法的对比总结

**与 best_fit 的区别**

- best_fit 只做页粒度（一次申请 n 页、一页就是最小单位），没有对象缓存：大量小对象会造成“整页只放一个小对象”的浪费，并且每次都要在全局链表里找最优块。
- SLUB 在页层仍用最佳适应，但在其上加了 size-class：同类小对象在 1 页内循环利用，拆 slab 后还能整页回收，因此显著降低小块外碎片、提高命中率。

**与 buddy_pmm 的区别**

- buddy 通过幂次拆分/合并让合并 O(logN)、数据结构简单，但是申请页块大小受 2k 限制，且需要频繁地切分页块。
- SLUB 仍然以“整页”为 slab，但能细粒度分配到对象尺寸，且每页内部自由链表避免了 buddy 系统常见的内部碎片；同时可接受任意长度（非幂次）页块，不再受 2k 限制；一个 slab 里的对象全部等大，避免了同大小的小规模内存申请时对页块进行频繁的划分。

- 代价是链表遍历和元数据维护相对复杂；目前代码没有实现 per-CPU 缓存或跨页 slab，所以在大对象场景下仍需其他分配器补充。

**总结** ：相较 best_fit，SLUB 在页层基础上扩展了对象缓存，极大改善小块管理；相较 buddy，它保持分块灵活并提供对象级接口。

## 5.5 扩展练习 Challenge：硬件的可用物理内存范围的获取方法

### 5.5.1 问题重述

如果 OS 无法提前知道当前硬件的可用物理内存范围，你有何办法让 OS 获取可用物理内存范围?

### 5.5.2 解决方案探讨

在理想情况下，操作系统（OS）内核并不需要“猜测”物理内存的布局。这项关键任务通常由更底层的软件——引导加载程序或固件来完成。这些底层软件对硬件有最直接的了解，它们负责在启动早期探测硬件，并将一份详细的**内存映射表**传递给操作系统内核。这是现代操作系统获取内存范围最标准、最可靠的方法。

根据不同的平台和启动标准，这种信息传递有多种形式：

- **x86 BIOS/E820：**在传统的 PC BIOS 系统中，bootloader 通过执行 `INT 15h, AX=E820h` 中断，从 BIOS 获取一个地址范围描述符结构列表。这个列表详细说明了物理地址空间中每一段的起始地址、长度和类型（如可用 RAM、ACPI 保留、硬件映射 IO 等）。GRUB 等 bootloader 会将这份 E820 表传递给 Linux 内核。
- **UEFI：**在现代的 UEFI 固件中，bootloader 通过调用 `GetMemoryMap()` 启动服务来获取内存映射。这份映射比 E820 更为详尽和规范。
- **设备树：**在 ARM 和 RISC-V 等嵌入式和服务器平台中，DTB 是描述硬件布局的标准。Bootloader（如 U-Boot 或本实验中的 OpenSBI）会将一个描述硬件信息的 DTB 文件加载到内存中，并将其地址传递给内核。内核启动后，一个核心任务就是解析这个 DTB，从中找到“/memory”节点，读取其“reg”属性，从而获知主内存（DRAM）的基地址和大小。这正是本实验 ucore 所采用的方法。

然而，如果面临一个极端情况，即没有任何来自底层软件的内存信息，操作系统只能尝试**自行探测**。这是一种复杂、充满风险且可靠性较低的“最后手段”。其基本原理是**试探性地读写内存**。

1. **确定探测起点：** OS 知道自身被加载到了哪个物理地址。探测可以从内核映像结束的位置（即 `_end` 符号）之后第一个页对齐的地址开始。
2. **执行探测循环：** OS 会以一个固定的步长（例如一个页或更大的块）向上遍历物理地址。
3. **“读-写-读-恢复”测试：**对于每一个待测试的地址，OS 会执行以下操作：

- a. 读取该地址的原始值并保存。
- b. 写入一个或多个特殊的“魔数”（例如 `0x55AA55AA`, `0xAA55AA55`），以确保不是巧合。
- c. 立即读回该地址的值，检查是否与写入的魔数一致。
- d. 如果一致，将原始值写回，以尽可能减少破坏。

4. **判断内存类型：**如果读回的值与写入的值完全一致，那么可以**高度怀疑**这个地址对应的是可读写的 RAM。如果不一致，则该地址可能对应 ROM、未映射的地址空间，或者是一个只写/只读的硬件寄存器（MMIO）。
5. **构建内存映射：**OS 根据探测结果，记录下所有被确认为 RAM 的连续地址范围，从而构建出自己的内存映射表。

**自行探测的巨大风险与局限性：**

- **危险性极高：**物理地址空间中不仅有 RAM，还散布着大量内存映射的设备 I/O 端口（MMIO）。向这些地址写入一个意外的值，可能会导致硬件设备进入未知状态，甚至直接导致系统崩溃或硬件损坏。
- **不完备性：**这种方法很难发现内存“空洞”（即两块可用 RAM 之间的非 RAM 区域）。
- **需要平台知识：**为了降低风险，一个进行自探测的 OS 通常仍需要一些关于目标平台架构的硬编码知识，例如知道哪些地址范围是绝对不能触碰的 MMIO 区域。
- **探测终点未知：** OS 不知道应该探测到多高的地址才算结束，通常只能设定一个不切实际的上限（如 4GB），或者在连续遇到大量非 RAM 地址后“猜测”内存已经结束。

综上所述，虽然理论上 OS 可以通过自行探测来获取内存范围，但在实践中，所有主流操作系统都依赖于 Bootloader/Firmware 提供准确的内存映射信息。这是一种清晰的、安全的、可靠的底层与上层软件之间的协作模式。

# 6 实验与理论对应

## 6.1 实验中出现相关知识点

- **分页内存管理：**实验核心，通过建立页表实现了虚拟地址到物理地址的映射，是课本分页管理机制的直接体现。
- **多级页表：**实验中使用了 Sv39 三级页表，对应了课本中为解决巨大页表空间问题而提出的多级页表方案。
- **TLB（快表）：**虽然我们没有直接编程操作 TLB，但 `sfence.vma` 指令的使用以及对地址翻译效率的讨论，都体现了 TLB 在硬件层面的重要性。
- **连续物理内存分配：** First-Fit 和 Best-Fit 算法是课本中介绍的经典可变分区分配策略。实验通过空闲链表的方式实现了这些算法。

- **内存碎片：**在分析 FF 和 BF 算法的优缺点时，必然会涉及到内部碎片和外部碎片的概念，这与理论知识完全吻合。

### 6.2 实验中未涉及相关知识点

- **分段管理：**实验直接基于分页，未实现段页式内存管理。RISC-V 架构本身也弱化了分段的概念。
- **页面置换算法：**本次实验只涉及物理内存的分配，不涉及虚拟内存的换入换出，因此没有实现如 FIFO, LRU, Clock 等页面置换算法。
- **请求分页：**实验中所有映射都是预先建立的，没有实现当发生缺页异常时才从外存加载页面的请求分页机制。
- **写时复制：**这是实现高效 `fork()` 的关键技术，在本次实验中未涉及。

## 7 实验所遇问题

### 7.1 问题描述

在实验初期进行环境配置和初步编译时，执行 `make qemu` 命令后，QEMU 无法正常启动 ucore 系统，程序在 OpenSBI 加载后即停止，没有任何内核输出，也无法进入内核执行流程。

### 7.2 问题解决

经过排查和咨询助教，最终定位问题是由于本地环境中使用的 OpenSBI 版本过高，与当前实验框架存在兼容性问题。实验框架所依赖的接口或行为在较新的 OpenSBI 版本中可能发生了变化，导致内核无法被正确引导。

解决方案是严格遵循实验指导的版本要求，具体步骤如下：

1. 重新下载并编译指定版本的 OpenSBI（版本 0.4）。
2. 修改项目根目录下的 `Makefile` 文件，将其中指向 OpenSBI 二进制文件的路径变量，改为指向新下载的 0.4 版本。
3. 清理旧的编译产物（`make clean`），然后重新执行 `make qemu`。

完成上述步骤后，系统成功启动，QEMU 窗口中可以观察到 ucore 内核的打印信息，问题得到解决。这次经历也说明了在进行底层系统开发时，严格匹配工具链和依赖库版本的重要性。