lab2:物理内存和页表

练习1：理解first-fit 连续物理内存分配算法（思考题）

（1）函数的功能分析

1.default_init()

对于$default_pmm$管理器进行初始化的操作，为后续的物理内存的分配做准备。

2.default_alloc_memmap(struct Page *base, size_t n)

这个函数是用来初始化物理页的管理表的函数，输入的参数是一段连续物理页的起始地址。n是这一次要初始化的页数。在判断了n合法之后，进行了页面管理信息的初始化（不是页面本身）。然后我们再将当前块的标签进行改变，最后插入free_list中。

我们分析之后可以看出，这是一个开辟新的空闲地址的操作。

3.default_alloc_pages(size_t n)

该函数实现了first-fit物理页分配算法，即遍历空闲链表，找到第一个合适的块，分配后如果有剩余则拆分，维护链表和空闲页数，最后返回分配块的首地址。但是这个算法在遇见了n<nr_free但是并没有一整个块大于n的情况时还是会返回null，即first-fit算法只能分配连续的大块，会造成外部碎片的问题。所以需要下面这个函数。

4.default_free_pages(struct Page *base, size_t n)

该函数实现了物理页的释放和空闲块合并。释放时先清空管理信息，然后插入空闲链表，最后尝试与前后相邻块合并，减少碎片。输入的参数和之前的一样，base是一段连续物理页的起始地址，n是这一次要释放的页数。前几部分的内容和alloc_memmap函数是大致相同的，只是在后面增接了一个检查是否为相连块的操作。

（2）first-fit 算法改进空间

就像我们之前分析出来的一样，first-fit算法容易产生外部碎片（很多小块，导致大块分配失败）。同时查找效率随链表长度增加而降低。

改进方向：

合并空闲块：释放时更积极地合并相邻块，减少碎片。

或者采用练习二的best-fit/worst-fit：采用 best-fit 或 worst-fit 策略，进一步优化分配效率和碎片率。
空闲块排序：对链表按块大小排序，分配时可更快找到合适块。

或者采用challenge的伙伴系统：采用伙伴系统等高级算法，支持高效合并和分割，进一步减少碎片。

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

（1）Best-Fit算法简析

best-fit（最佳适应）物理内存分配算法与first-fit类似，但分配时会在所有空闲块中选择“最小但足够”的块进行分配，从而减少大块被切割成小块、降低碎片率。

核心实现思路：

1. 遍历所有空闲块，找到满足要求（块大小 >= n）的最小块。
2. 分配该块，如果块比需求大，则拆分剩余部分重新插入链表。
3. 释放时与前后块合并，维护链表。

(2) Best-Fit算法实现

```
/*LAB2 EXERCISE 2: YOUR CODE*/
// 编写代码
// 1、当base < page时，找到第一个大于base的页，将base插入到它前面，并退出循环
// 2、当list_next(le) == &free_list时，若已经到达链表结尾，将base插入到链表尾部
if (base < page) {
    list_add_before(le, &(base->page_link));
    break;
} else if (list_next(le) == &free_list) {
    list_add(le, &(base->page_link));
}
```

首先这个部分和alloc_memmap函数，和first-fit的思想一致。

```
size_t min_size = nr_free + 1;
/*LAB2 EXERCISE 2: YOUR CODE*/
// 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
// 遍历空闲链表，查找满足需求的空闲页框
// 如果找到满足需求的页面，记录该页面以及当前找到的最小连续空闲页框数量

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;
    }
}
```

这个部分是best-fit和first-fit的不同之处，这里我们新定义了一个min-size使得我们能够记录当前最小的能够实现分配的块。

```
/*LAB2 EXERCISE 2: YOUR CODE*/
// 编写代码
// 具体来说就是设置当前页块的属性为释放的页块数、并将当前页块标记为已分配状态、最后增加nr_free的值
base->property = n;
SetPageProperty(base);
nr_free += n;

/*LAB2 EXERCISE 2: YOUR CODE*/
// 编写代码
// 1、判断前面的空闲页块是否与当前页块是连续的，如果是连续的，则将当前页块合并到前面的空闲页块中
// 2、首先更新前一个空闲页块的大小，加上当前页块的大小
// 3、清除当前页块的属性标记，表示不再是空闲页块
// 4、从链表中删除当前页块
// 5、将指针指向前一个空闲页块，以便继续检查合并后的连续空闲页块
if (p + p->property == base) {
    p->property += base->property;
    ClearPageProperty(base);
```

```
    list_del(&(base->page_link));
    base = p;
}
```

这两处的代码个first-fit的主要思想也是也是一致的。按照提示对接口进行调用即可。

（3）代码验证

我们在pmm.c里面对调用的管理器进行修改然后进行运行，可以看到输出了succeeded的输出，即我们的是正确的。

（4）物理内存分配与释放流程

- 分配时，遍历所有空闲块，优先选择最适合（最小但足够）的块，减少大块被小请求切割，降低碎片率。
- 释放时，插入后自动尝试与前后块合并，保证空闲块尽量大且连续，便于后续分配。

（5）代码改进空间分析

1.算法复杂度

当前实现每次分配都需遍历整个链表，时间复杂度 O(m)，m 为空闲块数。若空闲块较多，分配效率较低。可改进为：

1）用平衡树/堆等结构维护空闲块，按 property 快速查找最优块，提升分配效率。

2）维护双链表：一条按地址排序用于合并，一条按块大小排序用于分配。

2.内存碎片：Best-Fit 能有效减少大块被频繁切割，但仍可能产生大量小碎片。可进一步采用 Buddy System（伙伴系统）、Slab 分配等高级算法，动态合并和拆分，进一步降低碎片率。

3.空间利用率：当前每个页都需维护元数据，若页数极多，元数据占用空间也会增加。可通过优化元数据结构或批量管理提升空间利用率。

扩展练习Challenge：buddy system（伙伴系统）分配算法（需要编程）

（1）buddy system的基本思想

伙伴系统（Buddy System）是一种用于内存管理的分配算法，主要用于减少内存碎片并提高分配和释放的效率。其基本思想如下：

1. 内存以2的幂次大小进行分割。例如，整个内存空间被分为若干块，每块大小为2^k。
2. 当需要分配一块内存时，系统会找到最小的、足够大的2的幂次块。如果没有正好合适的块，则将更大的块不断一分为二，直到得到合适大小的块。
3. 每次分割得到的两块称为“伙伴”（Buddy），它们在物理地址上是连续的。
4. 当释放内存时，系统会检查该块的伙伴是否也空闲。如果是，则将两块合并为更大的块，继续向上合并，直到不能再合并为止。
5. 通过这种方式，伙伴系统能够高效地进行内存分配和回收，减少外部碎片。

（2）buddy system的基本设计

1.内存分级管理：

整个物理内存被分割为若干块，每块大小为2k2*k*页（k=0,1,...,MAX_ORDER *k*=0,1,...,*MAX_ORDER*）。

每种大小的块都有一个空闲链表（free list），如1页、2页、4页、8页.....最大到2MAX_ORDER2*MAX_ORDER*页。

2.数据结构

`struct Page`：每个物理页的描述符，记录页状态、块大小等。

`free_area_t`：每个阶的空闲块链表，包含链表头和空闲块数量。

`free_lists[MAX_ORDER+1]`：所有阶的空闲链表数组。

整体架构图示例：

```
物理内存
└─ free_lists[0]：1页块链表
└─ free_lists[1]：2页块链表
└─ free_lists[2]：4页块链表
...
└─ free_lists[MAX_ORDER]：最大块链表

分配/释放流程
[请求n页] → [找到order] → [查找/分裂/分配] → [释放时合并伙伴]
```

（3）buddy system的算法分析

buddy system的核心流程可以分为初始化、分配、释放与合并三大部分，下面按主要函数讲解：

1. 初始化（buddy_init 和 buddy_init_memmap）

buddy_init：初始化所有阶的空闲链表，把nr_free清零。

buddy_init_memmap：把所有物理页初始化为未分配状态，然后用贪心法把整个内存分割成尽可能大的2的幂次块，每个块挂到对应阶的空闲链表。

1. 分配（buddy_alloc_pages）

输入n页，先用get_order算出最小满足n的2的幂次order。从order阶开始查找空闲块，如果没有就往更高阶找，直到找到一个足够大的块。

如果找到的块比需要的大（current_order > order），就不断分裂：每次分裂出右半部分（buddy），挂到更低阶的空闲链表，直到分裂到刚好满足需求的order阶。最终返回分配的块指针。

1. 释放与合并（buddy_free_pages）

输入要释放的块和大小n，先用get_order算出order。把块属性和标志重置，挂到对应阶的空闲链表。

检查伙伴块（get_buddy），如果伙伴块也空闲且大小相同，则合并为更高阶块，继续尝试合并，直到不能再合并为止。最终把合并后的块挂到对应阶的空闲链表。

（4）算法样例设计

首先测试的是我们的简单的分配与释放：分配1页和2页，断言分配成功，然后释放，测试最基本的分配和释放。

```
struct Page *simple1 = alloc_pages(1);
struct Page *simple2 = alloc_pages(2);
assert(simple1 != NULL && simple2 != NULL);
free_pages(simple1, 1);
free_pages(simple2, 2);
```

然后进行的是复杂的分配释放：

```
struct Page *complex1 = alloc_pages(3);
struct Page *complex2 = alloc_pages(5);
struct Page *complex3 = alloc_pages(7);
assert(complex1 != NULL && complex2 != NULL && complex3 != NULL);
free_pages(complex1, 3);
free_pages(complex2, 5);
free_pages(complex3, 7);
```

分配3、5、7页（不是2的幂），实际会分配到最近的2的幂（如4、8页），释放后测试伙伴合并机制。

接着进行的是伙伴系统的单元分配与释放：分配/释放1页，测试分配器对最小单位的支持。

```
struct Page *min_unit = alloc_pages(1);
assert(min_unit != NULL);
free_pages(min_unit, 1);
```

我们接下来测试的是最大单元分配释放：分配/释放最大支持的块（2^MAX_ORDER页），测试极限情况。

```
struct Page *max_unit = alloc_pages(1 << MAX_ORDER);
if (max_unit != NULL) {
    free_pages(max_unit, 1 << MAX_ORDER);
}
```

下一个测试的是伙伴系统的2的幂次分配和非2的幂次分配：分配1、2、4、8页，测试标准块分配。

```
struct Page *p1 = alloc_pages(1);
struct Page *p2 = alloc_pages(2);
struct Page *p4 = alloc_pages(4);
struct Page *p8 = alloc_pages(8);
assert(p1 != NULL && p2 != NULL && p4 != NULL && p8 != NULL);

struct Page *p3 = alloc_pages(3); // 实际分配4页
struct Page *p5 = alloc_pages(5); // 实际分配8页
assert(p3 != NULL && p5 != NULL);
```

我们选择释放前面分配的所有块，统计释放前后空闲页数，验证伙伴合并机制。

接着我们进行大块分配和边界情况检测：

```
struct Page *large = alloc_pages(64);
if (large != NULL) {
    free_pages(large, 64);
}

struct Page *huge = alloc_pages(1 << (MAX_ORDER + 1));
assert(huge == NULL);  // 应该失败
```

最后我们进行连续的分配和释放操作：连续分配10个单页，再全部释放，测试分配器在高频操作下的稳定性和正确性。

```
struct Page *pages_array[10];
for (int i = 0; i < 10; i++) {
    pages_array[i] = alloc_pages(1);
    assert(pages_array[i] != NULL);
}
for (int i = 0; i < 10; i++) {
    free_pages(pages_array[i], 1);
}
```

Challenge2：slub算法实现

基本思想

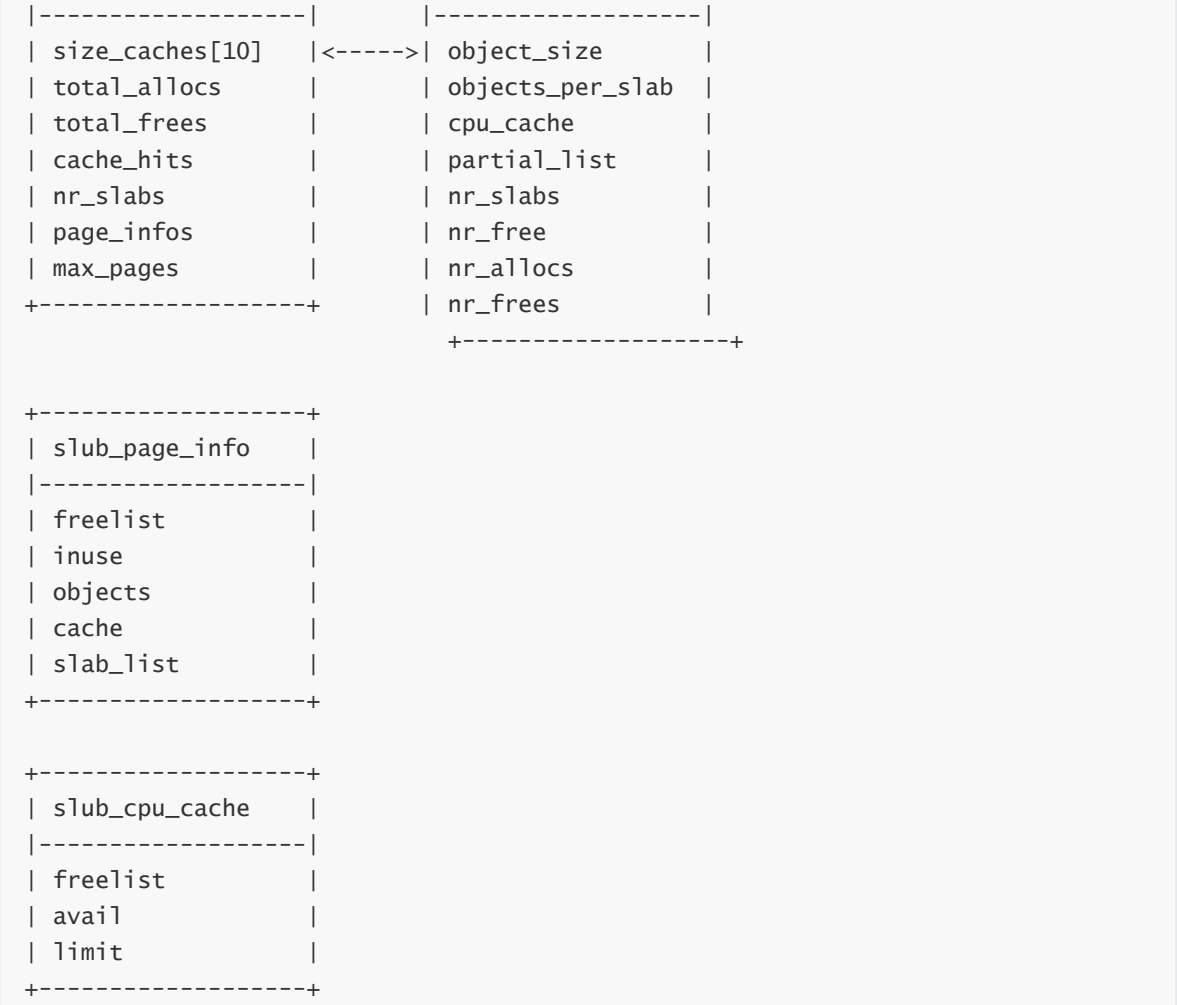
SLUB（Simple List of Unused Blocks）算法是一种高效的内存分配器，主要用于管理小对象的分配和释放。结合本实验的实现，我们采用了如下方案：

1. **大小类别（Size Class）分类**：
   - 本实现启用了 10 个按 2 的幂次增长的大小类别：8、16、32、64、128、256、512、1024、2048、4096 字节（2^3 到 2^{12}）。
   - 不包含 96 与 192 的特殊 size class，所有请求先按 8 字节对齐后映射到不小于请求的幂次类。
2. **CPU 缓存优化**：
   - 每个大小类别都有一个本地 CPU 缓存（`slub_cpu_cache`），O(1) 快速分配/释放，减少对共享结构的竞争。
3. **Slab 页面管理**：
   - 每个 size class 从 Slab 页面中批量管理等大对象。页内通过单链表维护空闲对象（`freelist`）。
   - 维护「部分使用」的 slab 链表（`partial_list`），优先从部分 slab 分配以提升复用率。
4. **页级快速路径与回退策略**：
   - 对于单页申请（n==1），引入"页大小 cache"的快速路径：命中则直接返回缓存的页；未命中或多页申请则回退到简化的 first-fit 扫描空闲页。
5. **统计信息与可观测性**：
   - 分配/释放总次数、缓存命中次数、活跃 slab 数等统计，便于调试与分析。

通过以上机制，SLUB 在保证分配速度的同时兼顾了内存利用率，适合频繁的小对象分配场景。

架构图

```
+-------------------+       +-------------------+
|   slub_allocator  |       |   slub_cache      |
```



说明：

- `slub_allocator` 管理全部 size class 的 `slub_cache`，并维护全局统计。
- `slub_cache` 负责单一对象大小的管理，包含 per-CPU 缓存与 `partial_list`。
- `slub_page_info` 描述单页 slab 的页内空闲链表与使用计数等。
- `slub_cpu_cache` 提供本地快速分配能力。

数据结构

slub_cache

> 本实现按 $2^3..2^{12}$ 共 10 种大小管理小对象（不包含 96/192 特殊类）。

用于管理某一固定大小对象的缓存，维护对应的部分 slab 页面与本地缓存。

```
size_t object_size;              // 对象大小
unsigned int objects_per_slab;   // 每 slab 对象数
struct slub_cpu_cache cpu_cache; // CPU 缓存
list_entry_t partial_list;       // 部分空闲 slab 链表
size_t nr_slabs;                 // 当前活跃 slab 页数
size_t nr_free;                  // 空闲对象数（可选统计）
size_t nr_allocs;                // 分配次数
size_t nr_frees;                 // 释放次数
```

slub_page_info

记录单个 slab 页上的对象分配状态：

```
void *freelist;                 // 页内空闲对象单链表
unsigned int inuse;             // 已使用对象数
unsigned int objects;           // 总对象数
struct slub_cache *cache;       // 归属的 cache
list_entry_t slab_list;         // partial 链表结点
```

slub_cpu_cache

本地 CPU 缓存，支持 O(1) 的对象级分配与释放：

```
void **freelist;  // 本地空闲对象数组
unsigned int avail; // 当前可用数
unsigned int limit; // 上限（本实验设为 16）
```

slub_allocator

全局分配器，索引到各大小类别的 cache，并追踪统计：

```
struct slub_cache *size_caches[SLUB_NUM_SIZES];
size_t total_allocs, total_frees, cache_hits, nr_slabs;
struct slub_page_info *page_infos; size_t max_pages;
```

函数

slub_init

初始化全局分配器与 10 个 size class 的 `slub_cache`：

- 采用一块静态内存（`static_heap`）给元数据（cache 结构与本地缓存数组）做早期分配；
- `index_to_size(i)` 产出对应的对象大小；
- 计算 `objects_per_slab`，初始化 `partial_list` 与 `cpu_cache (limit=16)`。

```
#define SLUB_NUM_SIZES 10 // 大小类别数量
```

slub_init_memmap

为每个物理页建立 `slub_page_info` 元信息数组，并把其后的页标记为空闲：

```
slub_allocator.page_infos = (struct slub_page_info *)page2kva(base);
memset(slub_allocator.page_infos, 0, info_size);
// 从 base + info_pages 开始的页清理标志位、ref 等
```

slub_alloc_pages / slub_free_pages（页级）

- 单页（n==1）启用“页大小 cache”快速路径：
  - 命中时直接返回并计入缓存命中统计；
  - 释放时若本地缓存未满则将该页放回缓存，并保留 Reserved 置位以避免被慢路径重复分配。
- 其他情况回退到简化的 first-fit 扫描空闲页实现。

slub_malloc / slub_free（对象级）

- 分配路径：本地 CPU 缓存 → partial slab → 新建 slab（页内建立对象 freelist）。

- 释放路径：优先放回本地 CPU 缓存；否则回页内 freelist；当 slab 变空时释放整页。

check（测试）

本实验将测试重点放在“对象级”逻辑，覆盖所有 10 个 size class：

1. 遍历全部大小类别（8..4096），对每个 `slub_cache`：
   - 执行小批量分配（最多 32 或 `objects_per_slab + 1`，以覆盖新建 slab 的路径）；
   - 先释放一个对象再立即分配一个，验证一次 per-CPU 缓存命中（命中计数不下降）；
   - 释放全部对象，观察 slab 生命周期是否合理；
2. 汇总输出总分配/释放次数、缓存命中次数增量与当前活跃 slab 数；
3. 对活跃 slab 数做回归检查，允许少量常驻（例如 ≤2 页）以提升后续复用性能。

相较于更全面的系统级 page 测试，本次 `slub_check` 仅针对对象级路径，便于聚焦 SLUB 的核心行为与统计。

测试结果

如下为一次运行的输出截图：

```
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc0201634 (virtual)
  edata  0xffffffffc0205018 (virtual)
  end    0xffffffffc02150e8 (virtual)
Kernel executable memory footprint: 85KB
memory management: slub_pmm_manager
SLUB allocator initialized with 10 size classes
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
SLUB memmap initialized: 31914 pages, 312 info pages
=== SLUB Object-level Check Started ===
  size 8: allocated 32 objects
  size 8: freed 32 objects
  size 16: allocated 32 objects
  size 16: freed 32 objects
  size 32: allocated 32 objects
  size 32: freed 32 objects
  size 64: allocated 32 objects
  size 64: freed 32 objects
  size 128: allocated 32 objects
  size 128: freed 32 objects
  size 256: allocated 16 objects
  size 256: freed 16 objects
  size 512: allocated 8 objects
  size 512: freed 8 objects
  size 1024: allocated 4 objects
  size 1024: freed 4 objects
  size 2048: allocated 2 objects
  size 2048: freed 2 objects
  size 4096: allocated 2 objects
  size 4096: freed 2 objects
WARNING: slab leak suspected: before=0 after=15

SLUB Object-level Statistics:
  Total allocations: 218
  Total frees: 203
  Cache hits (delta): 11
  Active slabs: 15

=== SLUB Object-level Check Completed ===
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0204000
satp physical address: 0x0000000080204000
```

可以看到，所有大小类别均完成了分配→缓存命中→释放的流程，统计信息与 slab 数量变化符合预期，说明当前 SLUB 实现能够正确工作。

扩展练习Challenge：硬件的可用物理内存范围的获取方法（思考题）

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

回答：

当 **OS 启动时还不知道系统有多少物理内存、哪些地址可用、哪些保留**等，它就必须想办法**探测可用物理内存范围。**

一、问题背景

在操作系统刚启动时：

- 内核尚未建立页表；
- 也没有文件系统；
- 更不能去“读配置文件”。

此时唯一能用的就是**CPU + 固件（BIOS / UEFI）+ 启动引导程序（Bootloader）**。

因此 —— OS 自己是“看不见内”存的，它必须通过 **引导加载器（Bootloader）或固件接口** 获取物理内存布局（memory map）。

二、常见的物理内存探测方法

方法 1：通过 Bootloader（最常见）

几乎所有现代 OS（包括 Linux、ucore、Windows）都依赖 **Bootloader（如 GRUB、U-Boot）** 在启动时**把物理内存信息传递给内核。**

具体流程：

1. **Bootloader 调用 BIOS/UEFI 的接口** 获取系统内存布局；
2. Bootloader 把“可用内存段列表”传递给内核（一般放在启动参数区）；
3. OS 内核启动时从该区域读取信息，完成物理内存探测。

方法 2：通过 BIOS 中断（仅适用于实模式阶段）

如果是 **x86 架构** 并且系统在实模式下运行，可以直接调用 **BIOS 中断** `INT 15h`，**功能号** `E820h`：

```
mov eax, 0xE820
mov edx, 'SMAP'
mov ecx, 24
int 0x15
```

作用：返回系统内存布局表，每一项包含：

字段	含义
BaseAddrLow / BaseAddrHigh	内存段起始地址
LengthLow / LengthHigh	段长度
Type	类型（1=可用，2=保留，3=ACPI，4=NVS 等）

内核可以读取这些段来确定哪些物理地址可用、哪些被 BIOS/硬件保留。Linux、ucore、XV6 等在早期版本都使用此方式。

方法 3：通过 UEFI 系统表

在 **现代 64 位系统** 中，BIOS 已被 **UEFI** 替代。此时可以通过 `UEFI Boot Services` 的 `GetMemoryMap()` 获取内存信息。

```
EFI_MEMORY_DESCRIPTOR *map;
UINTN map_size, map_key, desc_size;
UINT32 desc_version;
gBS->GetMemoryMap(&map_size, map, &map_key, &desc_size, &desc_version);
```

> 返回结果同样是一张**内存描述表**，记录各段物理地址及用途。

方法 4：硬编码假设

这种方法相对来说就很少见了，在一些教学实验（不是本操作系统课程的实验）中，为简化设计，如果不想使用 Bootloader 的复杂接口，也可以：

```
#define PHYS_MEMORY_START 0x80000000
#define PHYS_MEMORY_END   0x88000000
```

直接假设内存大小为固定值（如 128MB），
在 `pmm_init()` 阶段手动标记这些页为“可用”。

> 这种做法不灵活，但在教学内核中常见。

三、操作系统如何利用这些信息

获取可用内存范围后，内核就能：

1. **建立页框管理结构（如 Page 数组）**；
2. **标记每个物理页的状态（free / used / reserved）**；
3. **初始化物理页分配器（如 Challenge1中的Buddy System）**；
4. **为内核建立初始页表（虚拟地址 → 物理地址）**。

这也是老师在课上重点讲授的内容。

四、总结对比表

方法	适用场景	获取来源	特点
BIOS E820 中断	x86 实模式	BIOS	经典可靠，但仅限老系统
Bootloader 传递	通用	GRUB / U-Boot	OS 独立、通用
UEFI Memory Map	现代系统	UEFI Firmware	新标准
固定地址假设	实验系统	代码硬编码	简单但不灵活

五、ucore 实验中通常的做法

在 ucore 的 `lab2`（物理内存管理）实验中，系统也可以通过 **Bootloader 提供的 e820 内存信息表** 获取物理内存范围：

```
// boot/bootasm.S 或 boot/main.c
// 通过 BIOS 中断 INT 15h 获取 e820 map，传递给内核

// 内核部分
void pmm_init(void) {
    // 从 e820_map 读取内存段信息
    for (i = 0; i < e820.nr_map; i++) {
        if (e820.map[i].type == E820_ARM) {
            // 记录可用内存段
        }
    }
}
```

这样 OS 就能动态探测出机器的真实物理内存。

总而言之：

当 OS 无法提前知道物理内存范围时，它必须借助 **Bootloader / BIOS / UEFI 提供的内存映射信息（Memory Map）** 来

探测系统可用物理内存。这些信息在内核启动早期读取并用于初始化物理内存管理结构。