Lab2 物理内存和页表实验报告

小组成员：王昱 王梓丞 孟启轩 实验一解决了“启动”的问题，实验二要开始解决操作系统的物理内存管理问题。我们需要理解页表的建立和使用方法、物理内存的管理方法，以及页面分配算法。

知识点梳理

物理内存管理动机与技术

- 内存是一种稀缺资源，计算机必须管理内存以优化性能。本实验使用的主要内存管理技术包括分页存储技术和虚拟内存技术，其中分页是虚拟内存的基础。
- 直接将进程放入内存会造成碎片，影响内存利用率。分页技术通过将进程分割成小块存储，减少内存碎片。分页需要页表来跟踪代码在内存中的位置。
- 对于超大进程（如大型游戏），虚拟内存技术可以逻辑上扩展内存。基于局部性原理，进程的所有代码不必同时在内存中，虚拟地址和物理地址的映射通过页表实现。

项目主要文件结构

本次实验聚焦操作系统的物理内存管理，主要文件包括：

- **entry.S**：构建页表并初始化，调用内核初始化函数。
- **init.c**：进行内核初始化操作，包括内存管理初始化。
- **pmm.c**：实现物理内存管理函数，包括内存初始化、分配和释放。
- **default_pmm.c/best_fit_pmm.c**：实现内存管理相关的具体功能，使用链表和页面结构体。

以页为单位管理物理内存

- **页表项**：在Sv39中，一个页表项占8B，包含物理页号、保留位、脏位、访问位等。多个页表项组合成多级页表以扩展虚拟地址空间，解决内存浪费问题。
- **多级页表**：使用三级页表实现更大的虚拟地址空间，支持映射不同大小的页，以提高内存利用率。
- **页表基址**：使用satp寄存器设置页表的物理地址，并建立翻译机制。

物理内存探测

OpenSBI可以扫描物理内存，探测结果保存在DTB中。DRAM物理内存范围为[0x80000000, 0x88000000)，其中一部分被内核和OpenSBI占用。使用Page结构体描述物理页面的状态，以便管理内存使用情况。

设计实现过程

在实现过程中，我们首先完成了物理内存的初始化，然后实现了分页和虚拟内存技术，通过页表维护虚拟地址和物理地址的映射。在内存管理部分，设计了分配和释放函数，并对碎片问题进行了处理。此外，通过建立TLB来提高内存访问效率，最终实现了高效的物理内存管理机制。

练习1：理解First-Fit连续物理内存分配算法

`first-fit`连续物理内存分配算法作为物理内存分配的基础方法，需要理解其实现过程。通过分析`kern/mm/default_pmm.c`中的相关代码，特别是`default_init`、`default_init_memmap`、

`default_alloc_pages`、`default_free_pages`等函数，描述程序在进行物理内存分配的过程及各个函数的作用。

default_init 函数

```
static void
default_init(void) {
    list_init(&free_list); // 初始化空闲页面链表
    nr_free = 0;           // 设置空闲页面数量为0
}
```

- **作用**：初始化内存管理模块的全局数据结构。
- **实现**：初始化一个空的链表 `free_list`，用于存储所有空闲页面，将 `nr_free`（表示当前系统中空闲页面的数量）初始化为0。

default_init_memmap 函数

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(PageReserved(p)); // 确保页面已经被预留
        p->flags = p->property = 0; // 清除标志和属性
        set_page_ref(p, 0); // 设置页面引用计数为0
    }
    base->property = n; // 设置初始属性（页面数量）
    SetPageProperty(base); // 设置页面属性标志
    nr_free += n; // 更新空闲页面计数

    // 将页面加入到空闲列表
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link)); // 如果列表为空，直接加入
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) { // 找到合适的位置插入
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link)); // 插入到合适的位置
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link)); // 插入到链表末尾
            }
        }
    }
}
```

- **作用**：初始化物理内存映射，设置初始的空闲页面列表。

- **实现**：遍历给定范围内的所有页面，将它们标记为空闲，并加入到 `free_list` 链表中，更新 `nr_free` 计数。

default_alloc_pages 函数

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) { // 如果请求的页面数量超过空闲页面总数，返回NULL
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) { // 查找足够大的空闲块
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        // 更新空闲列表
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) { // 如果空闲块大于请求的页面数量，分割空闲块
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n; // 更新空闲页面计数
        ClearPageProperty(page); // 清除页面属性标志
    }
    return page;
}
```

- **作用**：分配连续的物理页面。
- **实现**：查找足够大的空闲块，分配页面，并更新空闲列表和计数器。

default_free_pages 函数

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) { // 标记页面为空闲
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
```

```
    }
    base->property = n; // 设置释放的页面数量
    SetPageProperty(base);
    nr_free += n; // 更新空闲页面计数

    // 加入到空闲列表
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link)); // 如果列表为空，直接加入
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) { // 找到合适的位置插入
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link)); // 插入到合适的位置
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link)); // 插入到链表末尾
            }
        }
    }

    // 合并相邻空闲块
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (p + p->property == base) { // 合并前一个块
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property == p) { // 合并后一个块
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
    }
}
```

- **作用**：释放之前分配的连续物理页面。
- **实现**：将释放的页面加入到空闲列表中，合并相邻的空闲块。

default_nr_free_pages 函数

```
static size_t
default_nr_free_pages(void) {
```

```
    return nr_free;
}
```

- **作用**：返回当前系统中空闲页面的总数。
- **实现**：直接返回全局变量 `nr_free` 的值。

问题：你的First-Fit算法是否有进一步的改进空间?

可以通过实现内存碎片合并机制来减少小空闲块的浪费，使用更高效的数据结构（如平衡树）来加快内存分配速度，动态调整分配策略以适应不同的内存使用模式，以及优化空闲块链表的管理。这些改进能够提高内存的利用率和分配效率，降低碎片化问题。

练习2：实现 Best-Fit 连续物理内存分配算法

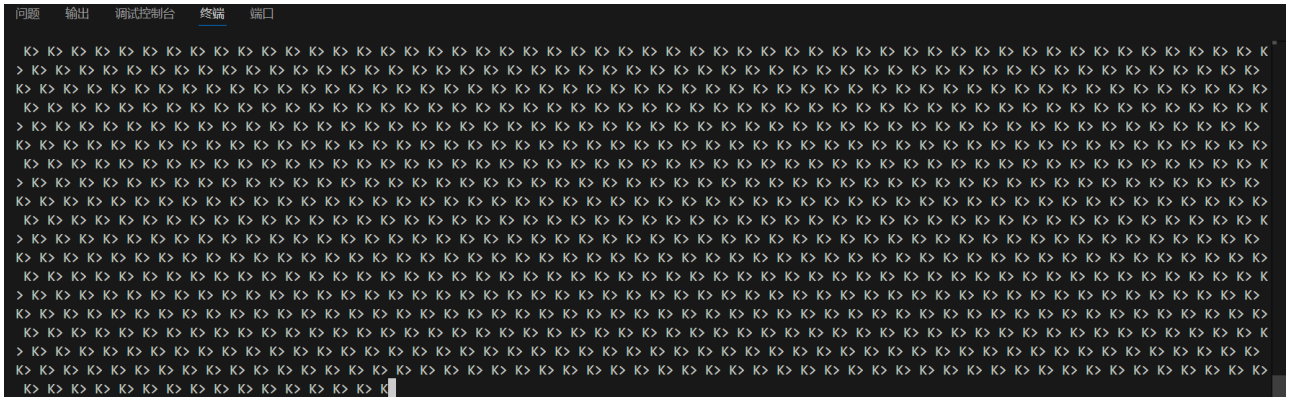
首先要理解Best-Fit算法的核心，与first-fit不同，best-fit算法会搜索整个空闲列表，找到能够满足请求的最小空闲块。

- 解决default_pmm.c中的重名问题后

```
free_area_t free_area_default;

#define free_list (free_area_default.free_list)
#define nr_free (free_area_default.nr_free)
```

- 调试输出发现以下结果



- 接下来对Best-Fit算法进行实现 主要对best_fit_alloc_pages函数做修改，遍历空闲列表，寻找最小空闲块，而不是找到空闲块就break。其余部分与First-Fit一致。

```
static struct Page *
best_fit_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1;
```

```
    /*LAB2 EXERCISE 2: 2212452*/
    // 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) { //在p->property >= n基础
上，还要是最小的
            page = p;
            min_size = p->property;
            //break;
        }
    }
```

- make qemu可以得到如下显示界面

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200032 (virtual)
  etext  0xffffffffc02019ae (virtual)
  edata  0xffffffffc0206010 (virtual)
  end    0xffffffffc0206470 (virtual)
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

```
Check PMM:              (2.3s)
  -check pmm:                               OK
  -check page table:                        OK
  -check ticks:                             OK
Total Score: 50/50
```

Best-fit算法通过维护一个空闲内存块的双向链表来寻找能够满足请求的最小空闲内存块，以减少内存浪费。在分配内存时，算法遍历链表找到第一个足够大的空闲块，如果找到的块比请求的大，则将其分割，并将剩余部分放回链表中。在释放内存时，算法将释放的内存块重新加入到链表中，并尝试与相邻的空闲块合并，以减少内存碎片。这个过程涉及到对链表的操作，包括插入、删除和合并节点，以及对页面属性的更新。Best-fit算法的目的是在满足内存请求的同时，尽可能地减少内存的浪费和碎片化。

你的 Best-Fit 算法是否有进一步的改进空间?

虽然Best-Fit算法试图减少内存浪费，但它通常不如First-Fit算法高效，因为它需要遍历整个空闲列表来找到最小的空闲块，这会增加内存分配的时间。此外，Best-Fit算法倾向于留下许多小的空闲块，可能会产生更多的外部碎片。 以下是可能的优化策略：

性能优化： Best-Fit算法在寻找最佳匹配的空闲块时可能需要遍历整个空闲列表，这在空闲块数量很多时会导致较高的查找成本。可以通过优化数据结构，例如使用平衡树或哈希表来存储空闲块，以提高查找效率。

内存碎片处理： 随着时间的推移，Best-Fit算法可能会产生越来越多的小内存碎片。可以引入内存压缩或合并机制，定期或在特定条件下对空闲列表进行整理，以减少碎片。

动态调整策略： 在某些情况下，Best-Fit可能不是最佳选择。可以设计一个动态调整的内存分配器，根据当前的内存使用情况和请求模式在Best-Fit和其他策略（如First-Fit、Worst-Fit或Next-Fit）之间切换。

局部性优化： Best-Fit算法通常不考虑内存访问的局部性原理。可以通过优化算法来优先分配那些物理上连续或接近的内存块给请求，从而提高缓存的效率。

实时监控与调整： 实现实时监控内存使用情况的功能，根据内存的使用趋势动态调整内存分配的阈值和参数，以适应不同的工作负载。

分页策略改进： 在多级页表系统中，可以考虑页面的大小和对齐方式，以减少页内偏移的开销，同时优化内存的分配和释放操作。

内存分配的公平性： 在多任务环境中，Best-Fit算法可能会导致某些进程获得过多的内存资源，而其他进程则资源不足。可以引入公平性机制，确保内存资源的合理分配。

内存分配的可预测性： 在某些实时或关键任务中，需要保证内存分配的可预测性。可以为这些任务预留内存资源，确保它们在任何时候都能获得所需的内存。

扩展练习Challenge：buddy system（伙伴系统）分配算法

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理, 每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

概述

Buddy 内存分配系统是一种高效的内存管理算法，能够快速分配和释放 2 的幂次方大小的内存块，并通过合并相邻的空闲块减少外部碎片。在操作系统中，内存管理至关重要，而 Buddy 系统能够有效地平衡内存分配效率与内存碎片问题。buddy算法将物理内存按照2的幂进行划分为若干空闲块，每次分配选择合适且最小的内存块进行分配。其优点是快速搜索合并(O(logN)时间复杂度)、低外部碎片（最佳适配best-fit），其缺点是内部碎片，因为空闲块一定是按照2的幂划分的。

设计目标

- **快速分配与回收**：通过二分法管理内存块，Buddy 系统能够以 O(logN) 的时间复杂度完成分配与回收操作。
- **减少外部碎片**：内存释放时，通过合并相邻的空闲块，最大限度减少外部碎片。
- **简单高效**：算法实现简单，易于理解与维护，并能高效地管理大块和小块内存。

3. 数据结构

```
typedef struct {
    list_entry_t free_list;  // 用于存储空闲块的双向链表
    unsigned int nr_free;    // 当前阶次的空闲块数量
} free_area_t;

struct Page {
    int ref;                 // 页框的引用计数
    uint64_t flags;          // 标记位，标记页面的状态
    unsigned int property;   // 页框的大小属性，表示该页框的大小
    list_entry_t page_link;  // 链接空闲块列表的指针
};
```

- free_area_t：每个阶次对应一个 free_area_t 结构，维护了一个链表用于存储该阶次的空闲块，以及一个 nr_free 用于记录当前空闲块的数量。
- Page：Page 结构体用来描述物理内存页框，包括页框的引用计数、标志位、块大小以及用于链接到链表的指针。

设计过程

- **buddy_init_memmap：** 初始化 base 所指的内存区域，将一块连续的物理内存标记为可用，按 2 的幂次方分配到相应的阶次中，并将其插入 free_area[order] 的空闲链表。确保初始化的内存页都是2的整数次幂。

```
static void buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    int order = MAX_ORDER;
    while ((1 << order) > n) {
        order--;
    }
    list_add(&free_area[order].free_list, &(base->page_link));
    base->property = (1 << order);
    SetPageProperty(base);
```

```
    free_area[order].nr_free++;
}
```

在系统启动或内存管理初始化阶段，操作系统会调用这个函数，将物理内存区域初始化为可分配的状态，使得 Buddy 内存分配算法可以高效管理和分配物理内存。

- **buddy_alloc_pages：** ：首先进行向上取2的阶，查找合适阶次 n 的内存块。如果当前阶次没有空闲块，则在更高阶次的块中查找；如果找到的是更大的块，那么就通过拆分操作将更大的块不断拆分，直到阶次为 n 。我们内部使用的仍然是first-fit的逻辑，从头遍历空闲链表，找到合适的空闲块，如果找到合适的块，则从空闲链表中删除该块并返回。

```
static struct Page *buddy_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > (1 << MAX_ORDER)) return NULL;

    int order = 0;
    while ((1 << order) < n) {
        order++;
    }
    // 查找适合的块
    for (int current_order = order; current_order <= MAX_ORDER; current_order++) {
        if (!list_empty(&free_area[current_order].free_list)) {
            list_entry_t *le = list_next(&free_area[current_order].free_list);
            list_del(le);
            free_area[current_order].nr_free--;
            struct Page *page = le2page(le, page_link);
            // 拆分较大的块，直到我们找到合适的大小
            while (current_order > order) {
                current_order--;
                // 创建一个 buddy 块
                struct Page *buddy = page + (1 << current_order);
                buddy->property = (1 << current_order);
                SetPageProperty(buddy);
                // 将 buddy 块放回空闲列表中
                list_add(&free_area[current_order].free_list, &(buddy-
>page_link));
                free_area[current_order].nr_free++;
            }
            // 拆分完成后，再从列表中删除该块
            list_del(le);
            free_area[current_order].nr_free--;
            ClearPageProperty(page);
            page->property = n;
            return page;
        }
    }
    return NULL;
}
```

- **buddy_free_pages：** 合并操作是减少外部碎片的关键，释放内存时，首先将其标记为空闲块。然后尝试与前后相邻的块合并。如果相邻的块与当前块大小相同且也未被占用，合并操作会继续，直到无法再合并为止。（其实也就是分配内存的逆操作）

```
static void buddy_free_pages(struct Page *base, size_t n) {
    int order = 0;
    while ((1 << order) < n) {
        order++;
    }

    base->property = (1 << order);
    SetPageProperty(base);

    while (order <= MAX_ORDER) {
        uintptr_t addr = page2pa(base);
        uintptr_t buddy_addr = addr ^ (1 << (PGSHIFT + order));
        struct Page *buddy = pa2page(buddy_addr);
        if (buddy_addr >= npage * PGSIZE || !PageProperty(buddy) || buddy-
>property != (1 << order)) {
            break;
        }
        list_del(&(buddy->page_link));
        ClearPageProperty(buddy);
        ClearPageProperty(base);
        base = (base < buddy) ? base : buddy;
        order++;
    }

    base->property = (1 << order);
    SetPageProperty(base);
    list_add(&free_area[order].free_list, &(base->page_link));
    free_area[order].nr_free++;
}
```

- **buddy_check：** 测试代码 buddy_check 用于验证 Buddy 系统的正确性。

```
static void buddy_check(void) {
    int count = 0, total = 0;
    // 遍历所有 order，检查每个阶次的空闲页块
    for (int order = 0; order <= MAX_ORDER; order++) {
        list_entry_t *le = &free_area[order].free_list;
        while ((le = list_next(le)) != &free_area[order].free_list) {
            struct Page *p = le2page(le, page_link);
            count++;
            total += p->property;  // 统计空闲块的总大小
        }
    }
    assert(total == buddy_nr_free_pages());
    struct Page *p0 = alloc_pages(26); //32- 32+
    assert(p0!= NULL);
```

```
    assert(!PageProperty(p0));
    assert((p0 + 32)->property == 32);
    struct Page *p1 = alloc_pages(26); //32- 32-
    free_pages(p0, 26);
    //这时候邻不是空闲状态就不会合并
    assert(p0->property==32); //32+ 32-
    free_pages(p1, 26);
    //这时候再放开就会合并了
    assert(p0->property==64); //64+
    basic_check();
}
```

这里首先测试了一下能否正确返回空闲页的大小，之后进行了一系列的合并释放操作，验证了代码可以正确进行分配释放操作，并且在邻块空闲的时候会进行合并操作,最后进行basic_check()验证代码是否可以完成基本检测。

接下来进行测试，如果没有提示错误信息，那么说明我们的buddy内存分配算法可以正确运行。下面是代码运行后的结果：

```
Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200032 (virtual)
  etext  0xffffffffc02016dc (virtual)
  edata  0xffffffffc0206010 (virtual)
  end    0xffffffffc0206560 (virtual)
Kernel executable memory footprint: 26KB
memory management: buddy_system_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087fffff
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
```

可以看见，并没有报错信息，也就是说我们的测试成功通过！

结论

Buddy 系统通过快速分配和高效的内存回收，适合用于操作系统中需要频繁分配大块和小块内存的场景。虽然有一定的内部碎片问题，但通过与其他内存分配机制结合使用（比如slub），Buddy 系统能在大多数场景下保持较高的内存利用率和良好的性能。

参考

伙伴分配器的一个极简实现

扩展练习Challenge：硬件的可用物理内存范围的获取方法

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？

通过 Device Tree (DT) 获取内存布局

在 RISC-V 和其他嵌入式架构中，操作系统常常通过 **设备树（Device Tree, DT）** 获取硬件资源的描述，包括可用的物理内存范围。设备树是一种硬件描述文件，它提供了关于系统硬件结构的关键信息，包括内存、CPU、总线、外设等。

设备树在系统引导过程中通常由引导加载程序（如 U-Boot）传递给操作系统。在 RISC-V 系统中，设备树的 `/memory` 节点通常包含物理内存的起始地址和大小。

通过 UEFI/ACPI 获取内存信息

RISC-V 还支持 **UEFI（统一可扩展固件接口）** 和 **ACPI（高级配置与电源接口）**，这些固件接口提供了获取物理内存布局的标准方式。

- **UEFI 内存映射**：类似于其他架构，RISC-V 的 UEFI 可以提供 `GetMemoryMap` 接口，让操作系统查询系统内存的物理布局。
- **ACPI**：虽然在 RISC-V 中 ACPI 还不是非常普及，但操作系统可以通过 ACPI 的表格（如 SRAT 或 E820 等）获取详细的内存映射。

优点：

- 这些标准化接口允许操作系统灵活获取内存布局。
- 已在 x86 和 ARM 架构上成熟应用，逐步在 RISC-V 中得到支持。

通过硬件探测（Memory Probe）

在某些情况下，RISC-V 系统可能没有设备树或 UEFI/ACPI 支持，操作系统可以通过硬件探测来判断内存范围。这种方法通常适用于简化的嵌入式系统。

- **内存探测机制**：操作系统可以尝试访问物理地址空间，检查内存区域是否可用。这通常通过向不同物理地址写入和读取数据来实现。
- **探测策略**：从较低的地址（如 0x80000000）开始逐步探测，直到遇到无效的内存访问或超出内存边界。探测到的有效地址范围将视为可用内存。

固件或引导加载器传递内存信息

在某些 RISC-V 系统中，操作系统的引导加载程序（如 U-Boot 或 OpenSBI）会在启动时将内存布局信息传递给内核。引导加载程序是第一个启动的固件，它可以识别系统内存的物理布局，并将这些信息通过某种机制传递给操作系统。

- **OpenSBI**：OpenSBI 是 RISC-V 的标准化引导固件，它可以与设备树结合，传递内存信息给操作系统。
- **U-Boot**：U-Boot 是广泛用于嵌入式系统的引导加载程序，它可以将设备树信息传递给操作系统内核。

通过特殊的硬件寄存器或 I/O 端口

在某些定制的 RISC-V 系统中，硬件可能通过特定的寄存器或 I/O 端口，直接提供内存布局信息。操作系统在启动时可以通过访问这些寄存器或端口，读取当前的物理内存配置。