

Lab2

一、实验目的

- 理解页表的建立和使用方法
- 理解物理内存的管理方法
- 理解页面分配算法

二、实验内容

实验一过后大家做出来了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。本次实验我们会了解如何发现系统中的物理内存，然后学习如何建立对物理内存的初步管理，即了解连续物理内存管理，最后掌握页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，帮助我们对段页式内存管理机制有一个比较全面的了解。本次的实验主要是在实验一的基础上完成物理内存管理，并建立一个最简单的页表映射。

三、实验过程

练习1：理解first-fit 连续物理内存分配算法（思考题）

1. 物理内存分配的过程以及各个函数的作用

- default_init：初始化空闲列表 free_list，并将空闲页面的数量 nr_free 设置为 0。
- default_init_memmap：初始化指定数量的页面。对于每个页面，设置其标志和属性，并将其添加到空闲列表中。如果空闲列表为空，则直接将新的页面块添加到空闲列表的开头。
- default_alloc_pages：函数从系统的空闲页面链表 free_list 中分配 n 个连续的页面，并将其从链表中移除，返回该页面块的起始地址。
- default_free_pages：释放指定数量的页面并将其重新插入到空闲列表中。在插入新页面时，检查是否可以合并相邻的空闲页面块，以减少碎片。
- default_nr_free_pages：返回当前空闲页面的数量。

2. first fit算法是否有进一步的改进空间

基于首次适应算法的内存管理系统虽然基本上是功能完备的，但仍然存在一些改进空间。以下是一些可能的改进方向：

- 并发访问：如果你的系统会在多线程环境中使用，考虑在分配和释放操作中引入锁机制，以避免数据竞争。同时，可以使用无锁数据结构来提高并发性能。
- 搜索优化：在大型系统中，遍历空闲列表可能会导致性能瓶颈。可以使用平衡树或其他高级数据结构来加速搜索过程，但建立平衡树需要付出一定的代价，需要综合考虑。
- 记录分配历史：添加一些功能来记录内存分配和释放的历史。这可以帮助进行调试和优化，尤其是在分析内存使用情况和性能时。
- 空间预留：在内存分配时预留额外的空间，以便未来的扩展或管理可能的内存碎片。

- 分层管理：将内存分为多个层次，每个层次有其特定大小的块，即使用十字链表类似结构来完成页块的存储。这种方式可以快速找到合适大小的块，减少搜索时间。

练习2：实现 Best-Fit 连续物理内存分配算法

(1) 代码编写

1. 在初始化函数，我们需要分配n个页作为一个页块，对于其中的每个页，均由结构体表示如下：

```

1 struct Page {
2     int ref;                                // page frame's reference counter
3     uint64_t flags;                          // array of flags that describe the
4     status of the page frame
5     unsigned int property;                  // the num of free block, used in f
6     irst fit pm manager
7     list_entry_t page_link;                // free list link
8 };

```

其中ref是其引用计数，flags是64位无符号整数，作为页面状态的标志位数组（位图），property用来表示当前空闲块的大小。特别是在合并相邻的连续空闲页面时，这个字段会记录块的大小。page_link是空闲页表的指针。

对于连续的n个页组成的空闲页块，我们只需要把第一个页设置property=n，且设置property标志位（与前述property含义不同），而对于后续所有页，**均清空当前页框的标志和属性信息，并将页框的引用计数设置为0即可**。

初始化n个页组成的空闲页块后，我们需要将其插入到空闲链表中，需要分为两种情况处理：

- 空表为空：

此时空表只有一个元素，其前向指针和后向指针均指向同一个地址（未知），直接让其后向指针指向该块即可（即后插）。

- 空表不为空：

由于链表的页存储按地址排序，故需要依次遍历，直到遇到大于该初始化块的地址的块，将其插入到这个块的前面，即可完成插入。如果遍历完整个链表均遇不到大于该地址的块，插入链表末尾即可。

完成代码如下：

```

1 static void
2 best_fit_init_memmap(struct Page *base, size_t n) {
3     assert(n > 0);
4     struct Page *p = base;
5     for (; p != base + n; p++) {
6         assert(PageReserved(p));
7
8         /*LAB2 EXERCISE 2: YOUR CODE*/
9         // 清空当前页框的标志和属性信息，并将页框的引用计数设置为0
10        p->flags = p->property = 0; // 页面的 flags 和 property 属性设为 0

```

```

11         set_page_ref(p, 0); // 将页面的引用计数设为 0。引用计数通常用于跟踪页
12         // 面是否正在被使用，0 表示页面是空闲的。
13     }
14     base->property = n;
15     SetPageProperty(base);
16     nr_free += n;
17     if (list_empty(&free_list)) {
18         list_add(&free_list, &(base->page_link));
19     } else {
20         list_entry_t* le = &free_list;
21         while ((le = list_next(le)) != &free_list) {
22             struct Page* page = le2page(le, page_link);
23             /* LAB2 EXERCISE 2: YOUR CODE */
24             // 编写代码
25             // 1、当base < page时，找到第一个大于base的页，将base插入到它
26             // 前面，并退出循环
27             // 2、当list_next(le) == &free_list时，若已经到达链表结尾，将
28             // base插入到链表尾部
29             if(base<page){
30                 list_add_before(le,&(base->page_link));
31                 break;
32             }
33         }
34     }
35 }
```

2. 分配空闲页面函数从系统的空闲页面链表 free_list 中分配 n 个连续的页面，并将其从链表中移除，返回该页面块的起始地址。

在first-fit中，是找到足够大小的空闲页表就分配，但是在best-fit中是想要找到比需要分配页表大小大但却是满足要求中最小的一块空闲页表。这种方式确保分配的内存块尽量接近请求大小，避免浪费更多的空闲内存。但是同样也存在缺点

- 遍历整个空闲链表：best-fit 必须遍历所有空闲的内存块才能找到最适合的块，因此在内存块较多时，查找时间可能较长，效率较低。
- 碎片问题：虽然在短期内减少了大块内存的浪费，但它会倾向于留下较小的内存碎片。过小的碎片以后可能难以使用。

代码如下：

```

1 static struct Page *
2 best_fit_alloc_pages(size_t n) {
3     assert(n > 0);
```

```

4   if (n > nr_free) {
5       return NULL;
6   }
7   struct Page *page = NULL;
8   list_entry_t *le = &free_list;
9   size_t min_size = nr_free + 1;
10  /*LAB2 EXERCISE 2: YOUR CODE*/
11  // 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
12  // 遍历空闲链表, 查找满足需求的空闲页框
13  // 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
14  while ((le = list_next(le)) != &free_list) {
15      struct Page *p = le2page(le, page_link);
16      if (p->property >= n&&p->property<min_size) {
17          min_size=p->property;
18          page = p;
19      }
20  }
21
22  if (page != NULL) {
23      list_entry_t* prev = list_prev(&(page->page_link));
24      list_del(&(page->page_link));
25      if (page->property > n) {
26          struct Page *p = page + n;
27          p->property = page->property - n;
28          SetPageProperty(p);
29          list_add(prev, &(p->page_link));
30      }
31      nr_free -= n;
32      ClearPageProperty(page);
33  }
34  return page;
35 }

```

3. 释放指定数量的页面并将其重新插入到空闲列表中。在插入新页面时，检查是否可以合并相邻的空闲页面块，以减少碎片。

Page结构体中flags前两位，一位表示reserved，一位表示property：

```

1 #define PG_reserved           0      // if this bit=1: the Page
2   is reserved for kernel, cannot be used in alloc/free_pages; otherwise,
3   this bit=0
4 #define PG_property          1      // if this bit=1: the Page
5   is the head page of a free memory block(contains some continuous_addr
6   ess pages), and can be used in alloc_pages; if this bit=0: if the Page
7   is the

```

因此对于重新变为空闲状态的块，需要将其块内第一个页的PG_property属性置位，即
SetPageProperty(base)

当释放指定数量的页面时，整个记录空闲页表的值也应该要相加释放的数量，然后在释放的时候要判断前后是否有空闲的页表能否与之合并，如果可以需要进行合并。

代码如下：

```
1 static void
2 best_fit_free_pages(struct Page *base, size_t n) {
3     assert(n > 0);
4     struct Page *p = base;
5     for (; p != base + n; p++) {
6         assert(!PageReserved(p) && !PageProperty(p));
7         p->flags = 0;
8         set_page_ref(p, 0);
9     }
10    /*LAB2 EXERCISE 2: YOUR CODE*/
11    // 编写代码
12    // 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
13    // 状态、最后增加nr_free的值
14    base->property=n;
15    SetPageProperty(base);
16    nr_free+=n;
17
18    if (list_empty(&free_list)) {
19        list_add(&free_list, &(base->page_link));
20    } else {
21        list_entry_t* le = &free_list;
22        while ((le = list_next(le)) != &free_list) {
23            struct Page* page = le2page(le, page_link);
24            if (base < page) {
25                list_add_before(le, &(base->page_link));
26                break;
27            } else if (list_next(le) == &free_list) {
28                list_add(le, &(base->page_link));
29            }
30        }
31
32        list_entry_t* le = list_prev(&(base->page_link));
33        if (le != &free_list) {
34            p = le2page(le, page_link);
35            /*LAB2 EXERCISE 2: YOUR CODE*/
36            // 编写代码
```

```

37 // 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
38 // 前页块合并到前面的空闲页块中
39 // 2、首先更新前一个空闲页块的大小，加上当前页块的大小
40 // 3、清除当前页块的属性标记，表示不再是空闲页块
41 // 4、从链表中删除当前页块
42 // 5、将指针指向下一个空闲页块，以便继续检查合并后的连续空闲页块
43 if((p+p->property)==base){
44     p->property += base->property;
45     ClearPageProperty(base);
46     list_del(&(base->page_link));
47     base=p;
48 }
49
50 le = list_next(&(base->page_link));
51 if (le != &free_list) {
52     p = le2page(le, page_link);
53     if (base + base->property == p) {
54         base->property += p->property;
55         ClearPageProperty(p);
56         list_del(&(p->page_link));
57     }
58 }
59 }

```

(2) 你的 Best-Fit 算法是否有进一步的改进空间？

- 首先与first-fit算法相似
- 由于best-fit算法在分配空闲块的时间复杂度会比first-fit代价高的情况，所以对于这两个算法的使用要根据自身情况判断

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

1.buddy_init_memmap

1. 首先，检查传入的页面数量 n 是否是 2 的幂。如果不是，则将 n 向下取整为最近的 2 的幂 (s)。
2. 设置 $buddy$ 的 $size$ 为计算出来的 2 的幂 s 。

$curr_free$ 被初始化为 s ，表示当前可用的空闲页数。

$longest$ 数组指向 $base$ 的内存地址。

$begin_page$ 初始化为 $base$ ，即内存页面的开始位置。

3. 二叉树的 `longest` 数组用于表示每个节点的空间块大小。从根节点开始，递归地计算和设置每个节点的块大小，依次从根节点的块大小开始，每次下探到子节点时，块大小减半，直到到达叶子节点。
4. 遍历 `begin_page` 到 `curr_free` 范围内的页面，将它们的属性重置为初始状态。
5. 将 `base` 的 `property` 设置为 `s`，代表有 `s` 个可用页面。

```

1  static void
2  buddy_init_memmap(struct Page *base, size_t n) {
3      cprintf("n: %d\n", n);
4      struct buddy *buddy = &b[id_++];
5      size_t s;
6      if(!IS_POWER_OF_2(n)){
7          s = last_power_of_2(n);
8      }
9      else{
10         s = n;
11     }
12
13
14     buddy->size = s;//buddy的大小
15     buddy->curr_free = s;//当前空闲的可用页数
16     buddy->longest = KADDR(page2pa(base));// 指向 base 的内存地址
17     buddy->begin_page = base;
18
19     size_t node_size = buddy->size * 2;
20
21     for (int i = 0; i < 2 * buddy->size - 1; i++) {
22         if (IS_POWER_OF_2(i + 1)) {
23             node_size /= 2;
24         }
25         buddy->longest[i] = node_size;
26     }
27
28     struct Page *p = buddy->begin_page;
29     for (; p != base + buddy->curr_free; p++) {
30         assert(PageReserved(p));
31         p->flags = p->property = 0;
32         set_page_ref(p, 0);
33     }
34     base->property = s;
35     SetPageProperty(base);
36 }
```

2.buddy_alloc_pages

1. 如果 n 不是 2 的幂，则使用 `next_power_of_2` 函数将其调整为下一个 2 的幂。
2. 遍历所有的 `buddy` 系统，找到一个满足页面请求的 `buddy`。它通过检查 `longest` 数组中是否存在一个块的大小大于或等于 n 。如果没有找到合适的 `buddy`，则返回 `NULL`，表示内存分配失败。
3. 从根节点开始，逐步查找合适的块，直到找到一个大小等于 n 的块。根据左子节点和右子节点的大小来决定分配哪个子块。
4. 将找到的块标记为已用（即将 `longest[index]` 设置为 0，表示该块不可用）。
5. 通过计算索引 `index` 和块大小 `node_size`，确定该块在整个 `buddy` 系统中的偏移量。
6. 向上递归更新父节点的状态，确保父节点的 `longest` 数组表示其两个子节点中较大的空闲块的大小。
7. 将 `buddy` 的 `curr_free` 数量减少 n ，表示系统中减少了 n 个空闲页面。
8. 返回分配的页面的起始地址。

```
1 static struct Page *buddy_alloc_pages(size_t n) {
2     // 确保请求的页面数量大于 0
3     assert(n > 0);
4
5     // 如果 n 不是 2 的幂，将其调整为下一个最接近的 2 的幂
6     if (!IS_POWER_OF_2(n))
7         n = next_power_of_2(n);
8
9     size_t index = 0;
10    size_t node_size;
11    size_t offset = 0;
12
13    struct buddy *buddy = NULL;
14
15    // 遍历所有的 buddy，找到一个可以满足页面请求的 buddy
16    for (int i = 0; i < id_; i++) {
17        if (b[i].longest[index] >= n) {
18            buddy = &b[i];
19            break;
20        }
21    }
22
23    // 如果没有找到合适的 buddy，则返回 NULL，表示分配失败
24    if (!buddy) {
25        return NULL;
26    }
27
28    // 在 buddy 的管理树中查找一个大小适中的块来分配
29    for (node_size = buddy->size; node_size != n; node_size /= 2) {
30        // 检查左子节点是否满足大小需求
```

```

31     if (buddy->longest[LEFT_LEAF(index)] >= n)
32         index = LEFT_LEAF(index);
33     // 否则检查右子节点
34     else
35         index = RIGHT_LEAF(index);
36     }
37
38     // 将找到的块标记为已用
39     buddy->longest[index] = 0;
40
41     // 计算该块在 buddy 的页面范围内的偏移量
42     offset = (index + 1) * node_size - buddy->size;
43
44     // 向上更新树的状态，确保父节点表示它的两个子节点中较大的空闲块
45     while (index) {
46         index = PARENT(index);
47         buddy->longest[index] = MAX(buddy->longest[LEFT_LEAF(index)], b
48         uddy->longest[RIGHT_LEAF(index)]);
49     }
50
51     // 减少 buddy 的当前空闲页面数
52     buddy->curr_free -= n;
53
54     // 返回分配的页面的起始地址
55     return buddy->begin_page + offset;
}

```

3.buddy_free_pages

1. 遍历所有的 `buddy` 系统，找到包含 `base` 的 `buddy` 系统。
2. 计算 `base` 相对于 `begin_page` 的偏移量 `offset`，以确定在 `buddy` 系统中的位置。
3. 根据 `offset` 计算出 `longest` 数组中的索引 `index`，以确定释放的块在二叉树中的位置。
4. 从叶子节点开始向上遍历，直到找到未被占用的节点（即 `longest[index] == 0`）。
5. 向上遍历更新父节点的块大小。如果左右子节点都为空闲块且其总大小等于当前父节点的块大小，则合并为一个更大的空闲块。否则，父节点的块大小更新为左右子节点中较大的那个。

```

1 static void
2 buddy_free_pages(struct Page *base, size_t n) {
3     struct buddy *bu = NULL;
4     for(int i = 0 ; i < id_ ; i++){//寻找base在哪个b[i]里
5         struct buddy *bb = &b[i];
6         if(base >= bb -> begin_page && base < bb -> begin_page + bb ->
size){

```

```
7         bu = bb;
8     }
9 }
10
11 unsigned node_size, index = 0;
12 unsigned left_longest, right_longest;
13 unsigned offset = base - bu->begin_page;
14
15 // 确保 self 指针有效, offset 在合法范围内
16 assert(bu && offset >= 0 && offset < bu->size);
17
18 // 初始化节点大小为1
19 node_size = 1;
20 // 计算当前节点在 longest 数组中的索引
21 index = offset + bu->size - 1;
22
23 // 从当前索引向上遍历父节点, 直到找到第一个未被占用的节点
24 for (; bu->longest[index]; index = PARENT(index)) {
25     // 每次向上遍历, 节点大小翻倍
26     node_size *= 2;
27     // 如果已经到达根节点, 退出循环
28     if (index == 0)
29         return;
30 }
31
32 // 在找到的空闲节点位置设置当前节点的大小
33 bu->longest[index] = node_size;
34 bu->curr_free += node_size;
35
36 // 更新父节点的大小信息
37 while (index) {
38     index = PARENT(index); // 移动到父节点
39     node_size *= 2; // 节点大小翻倍
40
41     // 获取左子节点和右子节点的最长空闲块大小
42     left_longest = bu->longest[LEFT_LEAF(index)];
43     right_longest = bu->longest[RIGHT_LEAF(index)];
44
45     // 如果左子节点和右子节点的大小和等于当前节点大小, 则更新父节点大小
46     if (left_longest + right_longest == node_size)
47         bu->longest[index] = node_size;
48     else
49         // 否则, 将父节点设置为左子节点和右子节点中的较大值
50         bu->longest[index] = MAX(left_longest, right_longest);
```

```
51     }
52 }
53 }
```

扩展练习Challenge：任意大小的内存单元slub分配算法 (需要编程)

1. 概述

SLUB (Simple List of Blocks) 是一种用于内核小块内存管理的分配算法，专门用于小块内存分配。SLUB 基于链表管理空闲内存块，在小块内存不足的情况下，能够通过大块内存的分配机制来保障内存供给。

该实现由两个主要文件组成：

- `slub.c`: 包含具体的 SLUB 分配算法实现，包括内存分配、释放及相关管理操作。
- `slub.h`: 声明了 SLUB 算法的核心接口。

2. 数据结构

2.1 `slob_block` 结构体

```
1 struct slob_block {
2     int units;
3     struct slob_block *next;
4 };
```

- `units`: 表示当前内存块大小，以 `SLOB_UNIT` 为单位。
- `next`: 指向链表中下一个空闲的内存块。

2.2 `bigblock` 结构体

```
1 struct bigblock {
2     int order;
3     void *pages;
4     struct bigblock *next;
5 };
```

- `order`: 指示分配的大块内存大小，通常是页大小的倍数。
- `pages`: 指向分配的大块内存区域的起始地址。
- `next`: 指向下一个 `bigblock` 结构体，形成一个链表。

2.3 静态变量

- `slob_t arena`: 用于初始化和管理小块内存的链表起始位置。
- `slob_t *slobfree`: 指向当前空闲的小块内存链表头。
- `bigblock_t *bigblocks`: 指向当前大块内存链表头，用于管理大块内存。

3. 宏定义

- `SLOB_UNIT`: 表示每个内存分配单元大小。
- `SLOB_UNITS(size)`: 计算分配所需的单元数，使内存分配的总大小不小于 `size`。

4. 函数设计

4.1 slub_init

```
1 | void slub_init(void);
```

初始化 SLUB 分配器，并输出初始化成功信息。

4.2 slob_alloc

```
1 | static void *slob_alloc(size_t size);
```

用于小块内存的分配操作：

1. 确保分配大小 `size` 小于页面大小 `PGSIZE`。
2. 遍历空闲链表，寻找合适的空闲块：
 - 若空闲块大小等于请求的大小，直接分配。
 - 若空闲块大小大于请求的大小，分割空闲块，将多余部分继续保留在空闲链表中。
3. 若无可用块，则申请新页面，并将其加入空闲链表。

4.3 slob_free

```
1 | static void slob_free(void *block, int size);
```

用于释放小块内存，回收到空闲链表中：

1. 确保 `block` 指针有效。
2. 计算块大小（单位为 `SLOB_UNIT`）。
3. 遍历空闲链表，插入并合并相邻块，确保空闲块链表结构的连续性。

4.4 slub_alloc

```
1 | void *slub_alloc(size_t size);
```

对外接口，分配指定大小的内存：

- 如果 `size` 小于一页大小，则调用 `slob_alloc` 分配小块内存。
- 如果 `size` 大于一页，调用 `alloc_pages` 分配大块内存，并将其记录在 `bigblock` 链表中。

4.5 `slub_free`

```
1 | void slub_free(void *block);
```

释放指定块的内存：

- 检查 `block` 是否为大块内存（通过地址是否对齐判断）。
- 对于大块内存，从 `bigblock` 链表中查找并释放。
- 对于小块内存，调用 `slob_free` 释放。

4.6 `slub_size`

```
1 | unsigned int slub_size(const void *block);
```

获取内存块的实际大小：

- 如果 `block` 为大块内存，则返回 `order << PGSHIFT`。
- 如果 `block` 为小块内存，则返回 `units * SLOB_UNIT`。

4.7 `slobfree_len`

```
1 | int slobfree_len();
```

计算并返回当前空闲的小块链表长度，用于调试和状态检查。

4.8 `slub_check`

```
1 | void slub_check();
```

测试和验证 SLUB 分配器的工作情况，依次进行分配和释放操作，输出空闲链表的状态。

5. 工作流程

1. **初始化**：调用 `slub_init` 初始化空闲链表。
2. **内存分配**：根据请求的 `size`，分配小块或大块内存，并维护对应的链表。
3. **内存释放**：根据 `block` 的类型，将内存块释放回空闲链表或大块链表。
4. **内存查询**：使用 `slub_size` 查询内存块大小，利用 `slobfree_len` 检查空闲链表长度。

6. 设计特点

- 采用链表管理空闲内存，减少内存碎片。
- 支持小块与大块内存分配，通过判断分配大小选择适合的内存管理策略。
- 代码结构清晰，便于扩展，适合内核环境中的小块内存管理。

7. 可能的优化

- 考虑将 `slob_alloc` 和 `slob_free` 中的内存合并逻辑进行优化，提高内存合并效率。
- 针对常用的小块分配场景，增加内存池加速常用大小的分配。

扩展练习Challenge：硬件的可用物理内存范围的获取方法 (思考题)

1. BIOS/UEFI 内存检测 (Memory Map)

1.1 使用 BIOS 中断调用 (在传统 BIOS 环境中)

在基于 BIOS 的系统中，操作系统可以通过调用 **INT 0x15, EAX=0xE820** BIOS 中断来获取内存映射信息。这是传统的方式，用于在启动时检测内存布局。

- **调用 INT 0x15, EAX=0xE820：**

- 通过此中断，操作系统可以获取内存范围列表，每个范围包含地址、大小以及该范围的类型（例如可用内存、保留内存、设备内存等）。
- 操作系统会循环调用这个中断，直到所有的物理内存区域被枚举出来。

例子：

```

1  struct memory_map_entry {
2      uint64_t base_addr; // 基地址
3      uint64_t length;   // 内存块长度
4      uint32_t type;    // 类型 (1 = 可用, 2 = 保留, 3 = ACPI Reclaim
5          able, 等等)
6      uint32_t reserved;
7  };
8
9  // 使用 INT 0x15, EAX=0xE820 获取内存映射
10 int get_memory_map(struct memory_map_entry *buffer, size_t buffer_si
11 ze) {
12     // 调用 BIOS 中断并填充内存映射表
13     ...
14 }
```

1.2 使用 UEFI 固件 (在现代 UEFI 环境中)

在现代的 UEFI (Unified Extensible Firmware Interface) 系统中，操作系统可以通过 UEFI 提供的 `GetMemoryMap` 函数获取内存映射。

- UEFI 的 `GetMemoryMap()` 函数返回一个描述符列表，每个描述符包含内存区域的类型、基地址和大小。
- UEFI 使得获取内存布局的过程更加简单，并且支持更多的内存区域类型，如 ACPI 数据区、内存映射的 I/O 区等。

例子：

```
1 EFI_STATUS GetMemoryMap(
2     UINTN *MemoryMapSize,
3     EFI_MEMORY_DESCRIPTOR *MemoryMap,
4     UINTN *MapKey,
5     UINTN *DescriptorSize,
6     UINT32 *DescriptorVersion
7 );
```

- 操作系统可以调用这个函数来获得当前系统的内存映射，并通过分析这些内存描述符来识别可用的物理内存范围。

2. 硬件抽象层 (例如 ACPI)

ACPI (Advanced Configuration and Power Interface) 是操作系统可以使用的另一种方式，通常在现代计算机系统中与 UEFI 一起使用。

- **ACPI 内存映射表**：操作系统可以通过解析 ACPI 表（例如 SRAT 和 E820）来获取系统内存布局。这些表会提供详细的系统硬件资源描述，包括可用内存的范围、CPU 结构等。
- **MADT 表**：用于描述系统中的多核 CPU 和其他资源分配。

ACPI 表通常由 BIOS 或 UEFI 提供，操作系统可以在启动时解析这些表来获取系统的内存布局。

3. 引导加载程序传递信息

引导加载程序（例如 GRUB 或其他自定义引导程序）通常会在启动时通过某种方式获取物理内存的信息，并将该信息传递给操作系统内核。

- **GRUB Bootloader**：如果使用 GRUB 作为引导加载程序，GRUB 在加载内核时，会将内存信息传递给内核。GRUB 通过多引导规范（Multiboot Specification）将内存映射传递给内核。

例子：

- 当操作系统符合 `Multiboot` 规范时，GRUB 会将物理内存的布局信息（通过 `e820` 获取）传递给内核。
- 操作系统通过读取 GRUB 传递的结构体，获取可用内存区域。

```
1 struct multiboot_memory_map_t {
2     uint32_t size;
3     uint64_t base_addr;
```

```
4     uint64_t length;
5     uint32_t type; // 类型 (1 = 可用, 2 = 保留, 等)
6 };
```

4. 物理地址空间探测 (内存探测)

如果以上方法不可用，操作系统可以尝试直接探测可用的物理内存。以下是几种物理地址空间探测方法：

- **写入/读取测试**：通过尝试向物理地址写入和读取数据，操作系统可以探测哪些内存地址是有效的。不过，这种方法非常低效，并且可能导致系统崩溃，尤其是在处理保留内存或设备内存时。因此这种方法在现代操作系统中不太推荐。

例子：

```
1 int is_memory_available(void *addr) {
2     volatile uint32_t *ptr = (volatile uint32_t *)addr;
3     uint32_t original = *ptr;
4     *ptr = 0xdeadbeef;
5     if (*ptr == 0xdeadbeef) {
6         *ptr = original;
7         return 1; // 可用
8     }
9     return 0; // 不可用
10 }
```

5. 通过特定硬件接口获取内存信息

某些特殊硬件平台上，系统固件或硬件接口提供了专门的方法来查询可用内存。例如，某些嵌入式平台上，系统启动固件会提供内存信息表或者通过总线接口获取内存信息。

总结

在设计操作系统时，以下几种方法可以帮助系统动态地获取可用物理内存的范围：

1. **通过 BIOS 中断 (INT 0x15, EAX=0xE820) 获取内存映射**，适用于传统的 BIOS 系统。
2. **通过 UEFI 的 GetMemoryMap 获取内存映射**，适用于现代 UEFI 系统。
3. **解析 ACPI 表**，特别是内存布局相关的表，例如 SRAT 和 E820。
4. **引导加载程序传递内存信息**，例如 GRUB 通过 **Multiboot** 规范传递内存布局。
5. **物理地址空间探测**，通过读取/写入内存地址进行探。

实验中的知识点

1. 最先匹配(First Fit)与最佳匹配(Best Fit)：

(1) 最先匹配(First Fit)：

思路：分配n个字节，使用第一个可用的空间比n大的空闲块。按分区的先后次序，从头查找，找到符合要求的第一个分区就分配。

示例：分配400字节，使用第一个1KB的空闲块。

分析：分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大

(2) 最佳匹配 (Best Fit)：

思路：分配n字节分区时，查找并使用不小于n的最小空闲分区。分区按小大顺序组织，找到的第一个适应分区是大小与要求相差最小的空闲分区。

示例：分配400字节，使用第3个空闲块(最小)

分析：个别来看，外碎片较小，整体来看，会形成较多外碎片。但较大的空闲分区可以被保留。

2. 分配算法的解耦实现：

在本次实验中，可以看到pmm_manager结构体中使用了多个字符或者函数指针，这些实际上是解耦和封装的实现方法，将两端的人的工作分离开来。

具体而言，提供结构体的人使用函数指针的形式将需要实现的函数分配算法的函数返回值和参数类型等格式规范告诉后端，后端只要实现其对应接口的函数即可；同样的，后端的人也不需要具体关注前端的人如何将他们实现的算法进行具体应用和调用的，而只需要专心关注于算法的设计和完善即可。

这使得在工程中两端不同的人只需要关注接口的实现，并不用关注互相的实现。大大方便了两端的人进行工程编程实现。

3. 页、页表和多级页表机制：

(1) 页 (Page)

页是物理内存和虚拟内存之间的一个固定大小的块。当程序需要更多的内存空间时，操作系统会为其分配一个或多个页。

页的大小通常是固定的，例如4KB、8KB等，具体大小取决于操作系统和硬件架构。

使用页的目的是为了使物理内存的管理更加高效，并支持虚拟内存技术。

(2) 页表 (Page Table)

页表是一个数据结构，用于存储虚拟页和物理页之间的映射关系。

当程序访问一个虚拟地址时，操作系统和硬件会使用页表来查找对应的物理地址。

页表通常存储在物理内存中，并由特定的硬件机制（如MMU，Memory Management Unit）进行管理和查找。

(3) 多级页表 (Multi-level Page Table)

由于现代计算机的内存容量非常大，单一的页表可能会非常庞大，从而占用大量的物理内存。为了解决这个问题，引入了多级页表机制。

在多级页表中，主页表只包含指向其他页表的指针，而这些子页表再指向更多的页表，如此递归，直到达到实际的物理页映射。

这种层次结构允许操作系统只加载当前活跃或正在使用的部分页表到内存中，从而节省内存。

例如，x86架构中的二级页表包括一个页目录和多个页表。而在某些架构中，如x86-64，存在四级页

表。

总的来说，页、页表和多级页表机制是现代计算机系统中虚拟内存管理的核心组件，它们允许程序认为它们拥有比实际物理内存更多的内存空间，并帮助操作系统更高效地管理物理内存。